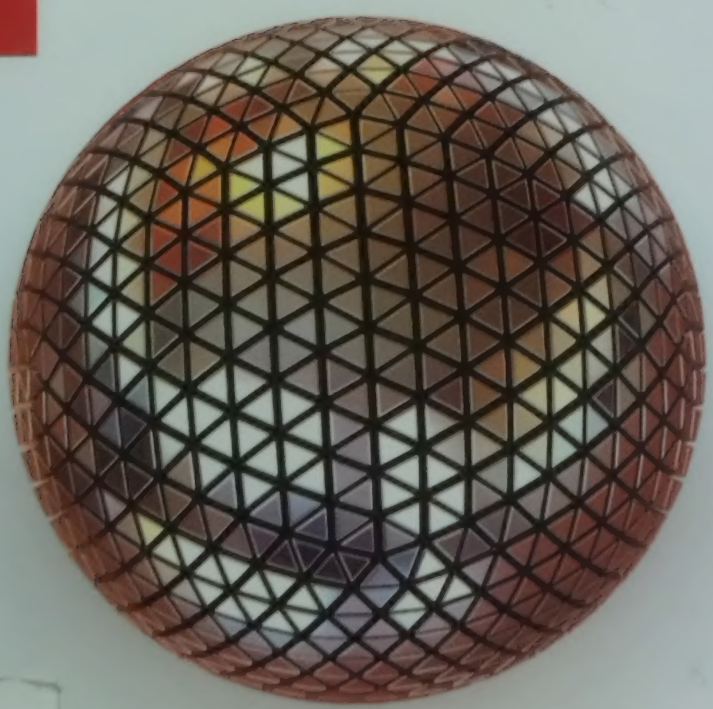
profesional
fundamente

Dorel Lucanu • Mitică Craus

FA

# PROIECTAREA ALGORITMILOR



III
7731

POLIROM

BIBL. CENTR. UNIV.
M. „EMINESCU" IAȘI
MATEMATICĂ
III-7131

III 7731

Dorel Lucanu, Mitică Craus

# Proiectarea algoritmilor

**Dorel Lucanu** a absolvit secţia de Informatică a Facultăţii de Matematică, Universitatea „Al.I. Cuza" Iaşi; este doctor în ştiinţe al Institutului de Matematică al Academiei Române. În prezent, este profesor universitar la Facultatea de Informatică, Universitatea „Al.I. Cuza" Iaşi, unde predă discipline de programare, algoritmi şi metode formale în ingineria software de la înfiinţarea acesteia (1992). Este laureat al premiului „Grigore Moisil" acordat de Academia Română.

**Mitică Craus** a absolvit secţia de Informatică a Facultăţii de Matematică, Universitatea „Al.I. Cuza" Iaşi; este doctor în ştiinţe al Facultăţii de Informatică din cadrul aceleiaşi universităţi. În prezent, este profesor universitar la Facultatea de Automatică şi Calculatoare, Universitatea Tehnică „Gheorghe Asachi" Iaşi, unde susţine cursuri de algoritmi secvenţiali şi paraleli, structuri de date, programare, limbaje formale şi translatoare din 1997.

© 2008 by Editura POLIROM

**www.polirom.ro**

Editura POLIROM
Iaşi, B-dul Carol I nr. 4; P.O. BOX 266, 700506
Bucureşti, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33, O.P. 37; P.O. BOX 1-728, 030174

**Descrierea CIP a Bibliotecii Naţionale a României:**

**LUCANU, DOREL**

**Proiectarea algoritmilor** / Dorel Lucanu, Mitică Craus – Iaşi: Polirom, 2008
Bibliogr.
Index

ISBN: 978-973-46-1140-9

I. Craus, Mitică

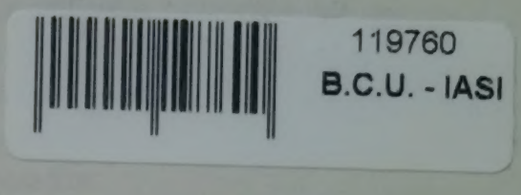
004.421
37.018.43

Printed in ROMANIA

III 7731

Dorel Lucanu • Mitică Craus

# PROIECTAREA ALGORITMILOR


119760
B.C.U. - IASI

POLIROM
2008

Dedicăm această carte familiilor noastre Geta, Tudor,
Diana și Thorsten, Irina și Adalbert-Christian.

# Cuprins


**Prefaţă** 9

**1 Despre algoritmi** 13
- 1.1 Limbaj algoritmic . . . 13
- 1.2 Probleme, algoritmi şi programe . . . 27
- 1.3 Măsurarea performanţelor unui algoritm . . . 30
- 1.4 Referinţe bibliografice . . . 40

**2 Tipuri de date de nivel înalt** 41
- 2.1 Liste liniare . . . 41
- 2.2 Liste liniare ordonate . . . 51
- 2.3 Liste circulare . . . 55
- 2.4 Stive . . . 57
- 2.5 Cozi . . . 62
- 2.6 Liste generalizate . . . 66
- 2.7 Arbori binari . . . 71
- 2.8 Coada cu priorităţi . . . 79
- 2.9 Grafuri . . . 83
- 2.10 *Union-find* . . . 101
- 2.11 Analiza amortizată . . . 104
- 2.12 Referinţe bibliografice . . . 108

**3 Derecursivare** 109
- 3.1 Parcurgerea în inordine a arborilor binari . . . 109
- 3.2 Recursia liniară . . . 113
- 3.3 Recursia în cascadă . . . 116
- 3.4 Memorarea într-un tabel a rezultatelor subproblemelor . . . 120
- 3.5 Referinţe bibliografice . . . 122

**4 Sortare internă** 123
- 4.1 Sortare bazată pe comparaţii . . . 124
- 4.2 Sortare prin distribuire . . . 138
- 4.3 Sortare prin numărare . . . 142
- 4.4 Sortare topologică . . . 144

4.5 Referinţe bibliografice . . . . . 146

**5 Căutare** **147**
5.1 Căutare în liste liniare . . . . . 148
5.2 Modelul de calcul al arborilor de decizie pentru căutare . . . . . 149
5.3 Arbori binari de căutare . . . . . 155

**6 Tipuri de date avansate pentru căutare** **161**
6.1 Arbori echilibraţi . . . . . 161
6.2 Dispersia (hashing) . . . . . 184
6.3 Arbori digitali (tries) . . . . . 190
6.4 Referinţe bibliografice . . . . . 198

**7 Căutare peste şiruri** **201**
7.1 Căutarea naivă . . . . . 202
7.2 Algoritmul Knuth-Morris-Pratt . . . . . 202
7.3 Algoritmul Boyer-Moore . . . . . 205
7.4 Algoritmul Rabin-Karp . . . . . 206
7.5 Expresii regulate . . . . . 208
7.6 Mai multe patternuri şi înlocuire . . . . . 211
7.7 Exerciţii . . . . . 214
7.8 Referinţe bibliografice . . . . . 215

**8 Despre paradigmele de proiectare** **217**
8.1 Aspecte generale . . . . . 217
8.2 Un exemplu simplu de paradigmă: eliminarea . . . . . 218
8.3 Alte consideraţii privind paradigmele de proiectare . . . . . 222
8.4 Referinţe bibliografice . . . . . 223

**9 Algoritmi *greedy*** **225**
9.1 Memorarea eficientă a programelor . . . . . 225
9.2 Prezentare intuitivă a paradigmei . . . . . 226
9.3 Arbori ponderaţi pe frontieră optimi . . . . . 227
9.4 Arbori Huffman . . . . . 232
9.5 Interclasare optimă pe două căi . . . . . 234
9.6 Problema rucsacului I (varianta continuă) . . . . . 236
9.7 Secvenţializarea optimă a activităţilor . . . . . 239
9.8 Problema instructorului de schi . . . . . 242
9.9 Arborele parţial de cost minim . . . . . 244
9.10 Prezentare formală a paradigmei . . . . . 246
9.11 Exerciţii . . . . . 248
9.12 Referinţe bibliografice . . . . . 251

**10 *Divide-et-impera*** **253**
10.1 Prezentare generală . . . . . 253
10.2 Sortare prin interclasare (*Merge Sort*) . . . . . 255
10.3 Sortarea rapidă (*Quick Sort*) . . . . . 258
10.4 Selecţionare . . . . . 262
10.5 Linia orizontului . . . . . 264
10.6 Transformata Fourier discretă . . . . . 270
10.7 Exerciţii . . . . . 274
10.8 Referinţe bibliografice . . . . . 275

**11 Programare dinamică** **277**
11.1 Exemplu: drum optim într-o reţea piramidală de numere . . . . . 277
11.2 Prezentarea intuitivă a paradigmei . . . . . 278
11.3 Alocarea resurselor . . . . . 279
11.4 Drumurile cele mai scurte între oricare două vârfuri ale unui digraf 282
11.5 Problema rucsacului II (varianta discretă) . . . . . 285
11.6 Subsecvenţa crescătoare maximală . . . . . 293
11.7 Distanţa între şiruri . . . . . 295
11.8 Arbori binari de căutare optimali . . . . . 298
11.9 Prezentarea formală a paradigmei . . . . . 301
11.10 Exerciţii . . . . . 304
11.11 Referinţe bibliografice . . . . . 308

**12 *Backtracking* şi *branch-and-bound*** **309**
12.1 Organizarea spaţiului soluţiilor candidat . . . . . 310
12.2 *Backtracking* . . . . . 319
12.3 *Branch-and-bound* . . . . . 320
12.4 Colorarea grafurilor . . . . . 325
12.5 Problema celor $n$ regine . . . . . 327
12.6 Submulţime de sumă dată . . . . . 330
12.7 Problema rucsacului II (continuare) . . . . . 332
12.8 Perspico . . . . . 337
12.9 Exerciţii . . . . . 340
12.10 Referinţe bibliografice . . . . . 342

**13 Probleme $\mathbb{NP}$-complete** **343**
13.1 Algoritmi nedeterminişti . . . . . 343
13.2 Clasele $\mathbb{P}$ şi $\mathbb{NP}$ . . . . . 344
13.3 Probleme $\mathbb{NP}$-complete . . . . . 348
13.4 Exerciţii . . . . . 354
13.5 Referinţe bibliografice . . . . . 354

**Bibliografie** **355**

**Index** **359**

# Prefaţă

Această carte are o perioadă de elaborare foarte lungă, greu de justificat, ce îşi găseşte originile în volumele doi şi trei ale cursului multiplicat [Luc96] şi ale cursului [CB02]. Cartea are la bază experienţa acumulată de-a lungul a optsprezece ani în cazul primului autor şi a doisprezece ani în cazul celui de-al doilea autor în predarea cursurilor de strucruri de date şi algoritmi, ce au purtat diverse denumiri de-al lungul anilor, la Facultatea de Informatică a Universităţii „Al. I. Cuza", respectiv la Facultatea de Automatizări şi Calculatoare a Universităţii Tehnice „Gh. Asachi", ambele din Iaşi.

## De ce o nouă carte despre algoritmi?

Studierea algoritmilor este o componentă obligatorie atât a programelor şcolare liceale, cât şi a programelor universitare ale liceelor şi facultăţilor ce includ clase, respectiv secţii cu profil de informatică. Ca atare, au apărut multe manuale şi cărţi despre algoritmi. În aceste condiţii se pune firesc întrebarea: ce poate aduce în plus sau original o nouă carte de algoritmi?

Asistăm în această perioadă la două fenomene majore: dezvoltarea tehnologică foarte rapidă şi accesul la un volum mare de informaţii. În fiecare zi apar situaţii în care trebuie să învăţăm din mers noi tehnologii, noţiuni sau concepte. Datorită grabei permanente, de multe ori suntem superficiali în înţelegerea acestora. Această superficialitate poate fi de înţeles atunci când dorim să ştim doar despre ce este vorba, eventual pentru a putea utiliza un nou produs sau o nouă tehnologie, dar nu este nici justificabilă şi nici scuzabilă atunci când dorim să ne formăm ca specialişti ai domeniului respectiv. Din păcate, mulţi autori de cărţi şi manuale, din dorinţa de a avea succes, au căzut în capcana acestui sistem de „învăţat din fugă". Cartea de faţă se doreşte a fi un contrabalans al acestei tendinţe. Gândirea algoritmică trebuie să fie un pilon de bază pentru oricine doreşte să devină un specialist în dezvoltarea de software (produse soft, programe de calculator). Gândirea algoritmică nu poate fi însuşită doar prin învăţarea unei colecţii de algoritmi; ea presupune însuşirea unui limbaj specific care trebuie să fie independent de limbajul de programare, abilitatea de a abstractiza pentru a vedea ce este esenţial în descrierea unui algoritm pentru a-l putea aplica în situaţii similare, cunoaşterea de instrumente prin care să se poată analiza algoritmul sau problema din punctul de vedere al corectitudinii, timpului de execuţie şi/sau a spaţiului de memorie necesar. Toate aceste aspecte le întâlnim în

MATEMATICA

toate etapele dezvoltării unei aplicaţii soft. Când spunem specialist în dezvoltarea de software, ne referim la cea ce se înţelege de fapt prin *inginer software*, care e diferit de un *tehnician software*: un tehnician software este capabil să scrie un program pe baza unor specificaţii date sau a unui algoritm dat; un inginer software utilizează în plus instrumente matematice cu ajutorul cărora proiectează soluţia şi dovedeşte că soluţia găsită este corectă şi performantă. Din acest punct de vedere, cartea de faţă a fost scrisă cu gândul la inginerul software.

Gândirea algoritmică este una interdisciplinară. Această interdisciplinaritate are două surse. Prima se referă la diversitatea domeniilor problemelor care necesită o soluţie algoritmică. A doua sursă este dată de partea de analiză a algoritmilor: aici vedem de ce este util să stăpânim instrumente învăţate la logică, grafuri, algebră, analiză matematică, probabilităţi etc. Credem că această interdisciplinaritate este unul dintre elementele principale care dau frumuseţe acestui domeniu.

Sumarizând, considerăm că am scris această carte cu gândul de a fi utilă pentru însuşirea pe îndelete a unei gândiri algoritmice, care să permită descoperirea frumuseţii acestei discipline, utilizând instrumentele cele mai potrivite.

## Cui se adresează această carte?

În termeni generali, răspunsul la această întrebare este dat de secţiunea precedentă. Aici încercăm să identificăm câteva situaţii concrete în care această carte este utilă.

În primul rând, cartea poate fi utilizată ca suport sau ca material consultativ la cursurile despre structuri de date şi algoritmi de la facultăţile sau secţiile cu profil informatic, atât de la universităţile clasice, cât şi de la cele cu profil tehnic.

Elevii de liceu găsesc în această carte multe dintre temele discutate la orele de informatică, descrise într-o manieră mai riguroasă şi cu informaţii suplimentare sau complementare. Multe dintre problemele de la concursurile de informatică îşi au sursa de inspiraţie în paradigmele prezentate aici.

Pentru profesorii de liceu care predau disciplina informatică, cartea poate constitui o sursă de inspiraţie pentru orele de predare sau un suport pentru perfecţionare.

Orice specialist în dezvoltarea de software poate găsi în această carte o sursă de inspiraţie pentru găsirea de soluţii la probleme concrete.

## Ce nu se găseşte în această carte...

Lumea algoritmilor este una foarte complexă şi cuprinzătoare, a cărei descriere nu poate fi inclusă într-o singură carte, fie ea şi de natură enciclopedică aşa cum este [Kao08]. Deoarece punem un accent deosebit pe conceptul de paradigmă de proiectare a algoritmilor, în sensul de model de gândire algoritmică, următoarele aspecte, importante în studiul algoritmilor, nu au putut fi incluse în prezenta carte.

Unul dintre domeniile cele mai prolifice pentru gândirea algoritmilor este teoria grafurilor şi combinatorica. Deşi cartea include multe exemple sau studii de caz din acest domeniu, nu am inclus un capitol dedicat algoritmilor specifici, deoarece considerăm că el constituie, de obicei, o disciplină separată.

Problemele NP-dificile şi problemele NP-complete sunt prezentate succint. Nu există un capitol special dedicat euristicilor şi algoritmilor de aproximare pentru aceste probleme. Este în intenţia autorilor de a elabora, eventual, în colaborare cu alţi colegi, o continuare a prezentei cărţi cu alte paradigme de algoritmi: probabilişti, genetici, paraleli, distribuiţi, ... .

Analiza algoritmilor presupune studierea şi a altor clase de complexitate care includ probleme practice de mare importanţă. Prezentarea acestora nu intră în scopul prezentei cărţi.

## Mulţumiri

Apariţia unei cărţi este rezultatul muncii mai multor persoane, chiar dacă pe copertă apar scrise numele a numai doi autori. Multe dintre secţiunile acestei cărţi sunt rodul discuţiilor cu domnii profesori Cornelius Croitoru şi Corneliu Bârsan (în prezent senior SDE la Microsoft). Capitolul despre structuri de date a căpătat actualul contur după multe discuţii cu Simona Orzan (actualmente la Universitatea din Eindhoven). Forma finală a soluţiilor unor studii de caz de la paradigme este rezultatul discuţiilor avute cu Laurenţiu Iancu (în prezent la Microsoft). Încurajările date de către domnii profesori Dan Cristea, Henri Luchian şi Sabin Buraga ne-au fost de mare ajutor.

Dacă există acum mai puţine greşeli, se datorează şi foştilor noştri studenţi Dana Buzdugan, Laurenţiu Vornicu, Ciprian Bacalu, Georgiana Caltais, Eugen Goriac, Ionuţ Gavrilă şi Alexandru Archip.

# Capitolul 1

# Despre algoritmi

## 1.1 Limbaj algoritmic

### 1.1.1 Introducere

Un *algoritm* este o secvenţă finită de operaţii (paşi) care, atunci când este executată, produce o soluţie corectă pentru o problemă precizată. Tipul operaţiilor şi ordinea lor în secvenţă respectă o logică specifică.

Algoritmii pot fi descrişi în orice limbaj, pornind de la limbajul natural până la limbajul nativ al unui calculator specific. Un limbaj al cărui scop unic este cel de a descrie algoritmi se numeşte *limbaj algoritmic*. Limbajele de programare sunt exemple de limbaje algoritmice.

În această secţiune descriem limbajul algoritmic utilizat în această carte. Limbajul nostru este tipizat, în sensul că datele sunt organizate în tipuri de date. Un *tip de date* constă dintr-o mulţime de entităţi (componente) de tip dată (informaţie reprezentabilă în memoria unui calculator), numită şi *domeniul* tipului, şi o mulţime de operaţii peste aceste entităţi. Convenim să grupăm tipurile de date în trei categorii:

- *tipuri de date elementare*, în care entităţile sunt indivizibile;
- *tipuri de date structurate de nivel jos*, în care entităţile sunt structuri relativ simple obţinute prin asamblarea de date elementare sau date structurate, iar operaţiile sunt definite la nivel de componentă;
- *tipuri de date structurate de nivel înalt*, în care componentele sunt structuri mai complexe, iar operaţiile sunt implementate de algoritmi proiectaţi de către utilizatori.

Primele două categorii sunt dependente de limbaj şi de aceea descrierile lor sunt incluse în această secţiune. Tipurile de nivel înalt pot fi descrise într-o manieră independentă de limbaj şi descrierile lor sunt incluse în capitolul 2. Un tip de date descris într-o manieră independentă de reprezentarea valorilor şi implementarea operaţiilor se numeşte *tip de date abstract*.

Paşii unui algoritm şi ordinea logică a acestora sunt descrise cu ajutorul *instrucţiunilor*. O secvenţă de instrucţiuni care acţionează asupra unor structuri de date

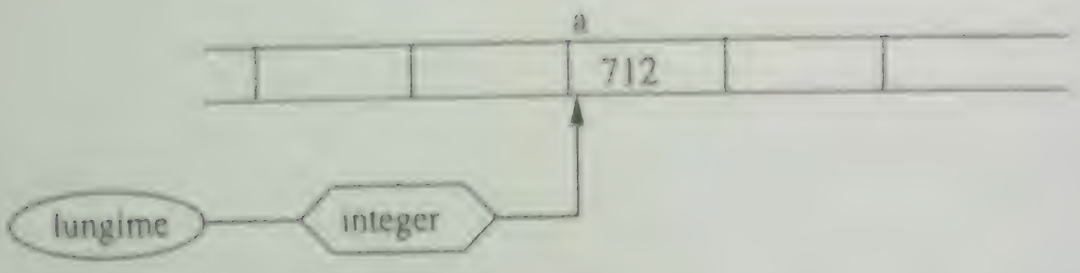
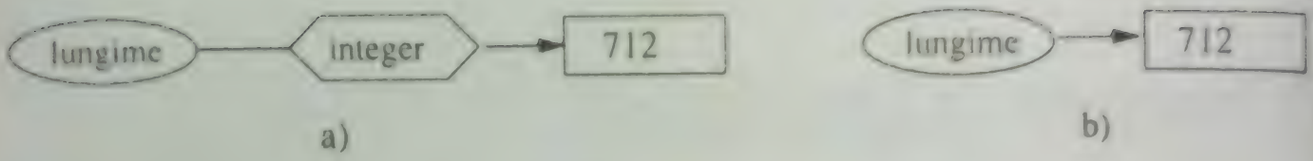



Figura 1.1: Memoria




a) b)

Figura 1.2: Variabilă

precizate se numeşte *program*. În secţiunea 1.2 vom vedea care sunt condiţiile pe care trebuie să le îndeplinească un program pentru a descrie un algoritm.

### 1.1.2 Modelarea memoriei

Memoria este reprezentată ca o secvenţă de celule (locaţii), fiecare celulă având asociată o *adresă* şi putând memora (stoca) o dată de un anumit tip (figura 1.1). Accesul la memorie este realizat cu ajutorul variabilelor. O *variabilă* este caracterizată de:

- un *nume* cu ajutorul căruia variabila este referită,
- o *adresă* care desemnează o locaţie de memorie şi
- un *tip de date* care descrie natura datelor memorate în locaţia de memorie asociată variabilei.

Dacă în plus adăugăm şi data memorată la un moment dat în locaţie, atunci obţinem o *instanţă a variabilei*. O variabilă este reprezentată grafic ca în figura 1.2 a. Atunci când tipul se subînţelege din context, vom utiliza reprezentarea scurtă sugerată în 1.2.b. Convenim să utilizăm fontul `type writer` pentru notarea variabilelor şi fontul *mathnormal* pentru notarea valorilor memorate de variabile.

### 1.1.3 Tipuri de date elementare

**Numere întregi.** Valorile sunt numere întregi, iar operaţiile sunt cele uzuale: adunarea ($+$), înmulţirea ($*$), scăderea ($-$) etc.

**Numere reale.** Deoarece prin dată înţelegem o entitate de informaţie reprezentabilă în memoria calculatorului, domeniul tipului numerelor reale este restrâns la submulţimea numerelor raţionale. Operaţiile sunt cele uzuale.

**Valori booleene.** Domeniul include numai două valori: `true` şi `false`. Peste aceste valori sunt definite operaţiile logice `and`, `or` şi `not` cu semnificaţiile cunoscute.

**Caractere.** Domeniul include litere: `'a'`, `'b'`, ..., `'A'`, `'B'`, ..., cifre: `'0'`, `'1'`, ... şi caractere speciale: `'+'`, `'*'`, .... Nu există operaţii.

**Pointeri.** Domeniul unui tip pointer constă din adrese de variabile aparţinând altui tip. Presupunem existenţa valorii NULL care nu referă nici o variabilă: cu ajutorul ei putem testa dacă un pointer referă sau nu o variabilă. Nu considerăm operaţii peste aceste adrese. Cu ajutorul unei variabile pointer, numită pe scurt şi pointer, se realizează referirea indirectă a unei locaţii de memorie. Un pointer este reprezentat grafic ca în figura 1.3.a. Instanţa variabilei pointer `p` are ca valoare adresa unei variabile de tip întreg. Am notat cu `integer*` tipul pointer al cărui domeniu este format din adrese de variabile de tip întreg. Această convenţie este extinsă la toate tipurile. Variabila referită de `p` este notată cu `*p`. În figurile 1.3.b şi 1.3.c sunt date reprezentările grafice simplificate ale pointerilor.

Pointerii sunt utilizaţi la manipularea variabilelor dinamice. O *variabilă dinamică* este o variabilă care poate fi creată şi distrusă în timpul execuţiei programului. Crearea unei variabile dinamice se face cu ajutorul instrucţiunii `new`. De exemplu, `new(p)` are ca efect crearea variabilei `*p`. Distrugerea (eliberarea spaţiului de memorie) variabilei `*p` se face cu ajutorul instrucţiunii `delete(p)`.

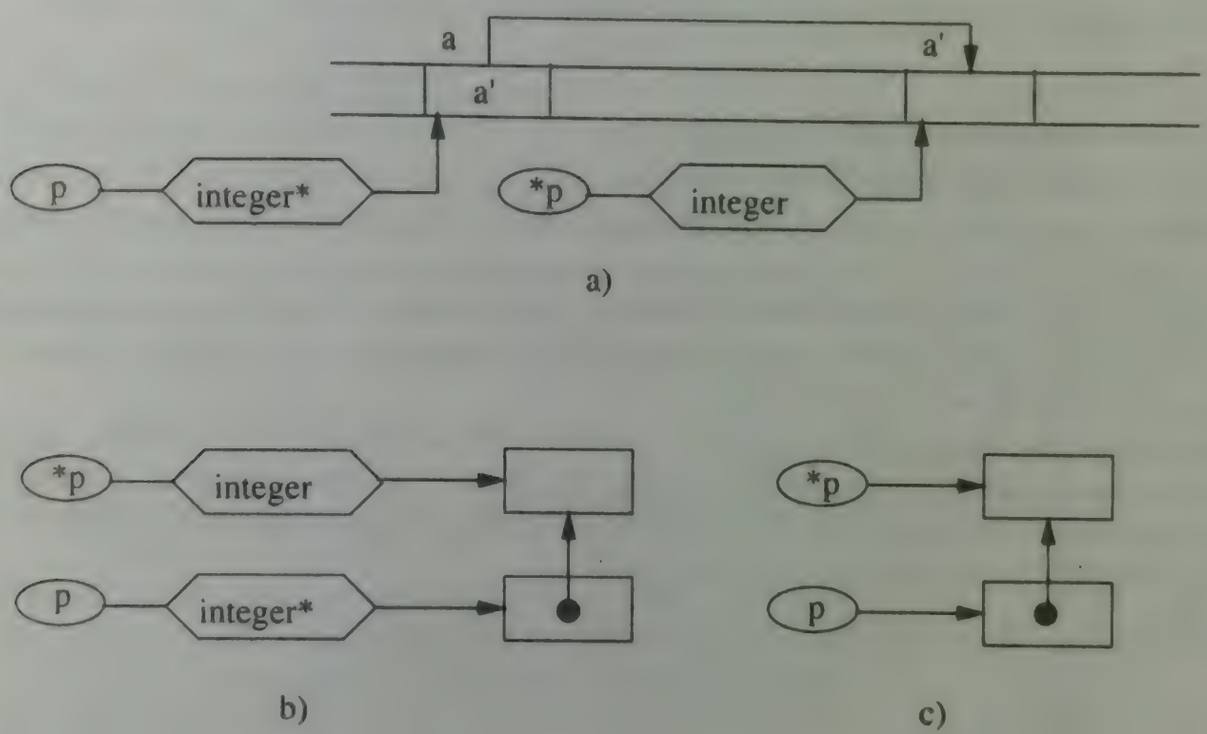



Figura 1.3: Pointer

### 1.1.4 Instrucţiuni

**Atribuirea.** *Sintaxa:*

⟨variabilă⟩ ← ⟨expresie⟩

unde ⟨variabilă⟩ este numele unei variabile, iar ⟨expresie⟩ este o expresie corect formată, de acelaşi tip cu ⟨variabilă⟩.

*Semantica*: Se evaluează ⟨expresie⟩ şi rezultatul obţinut se memorează în locaţia de memorie desemnată de ⟨variabilă⟩. Valorile tuturor celorlalte variabile rămân neschimbate. Atribuirea este singura instrucţiune cu ajutorul căreia se poate modifică memoria. O reprezentare intuitivă a efectului instrucţiunii de atribuire este dată în figura 1.4.

a) Inainte de atribuire

b) Dupa atribuirea "a <- a*b"

Figura 1.4: Atribuirea

**if.** *Sintaxa*:

```
if ⟨expresie⟩
   then ⟨secvenţă-instrucţiuni1⟩
   else ⟨secvenţă-instrucţiuni2⟩
```

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean, iar ⟨secvenţă-instrucţiuni$_i$⟩, $i = 1, 2$, sunt secvenţe de instrucţiuni scrise una sub alta şi aliniate corespunzător. Partea `else` este facultativă. Dacă partea `else` lipseşte şi ⟨secvenţă-instrucţiuni$_1$⟩ este formată dintr-o singură instrucţiune, atunci instrucţiunea `if` poate fi scrisă şi pe un singur rând.

*Semantica*: Se evaluează ⟨expresie⟩. Dacă rezultatul evaluării este `true`, atunci se execută ⟨secvenţă-instrucţiuni$_1$⟩, după care execuţia instrucţiunii `if` se termină; dacă rezultatul evaluării este `false`, atunci se execută ⟨secvenţă-instrucţiuni$_2$⟩ după care execuţia instrucţiunii `if` se termină.

**Observaţie.** O instrucţiune `if` de forma:

```
if (e1)
   then i1
   else if (e2)
           then i2
           else if (e3)
                   then i3
                   else if (e4)
                           ...
                           else in
```

va fi scrisă mai simplu astfel:

```
switch
   case (e1)
        i1
   case (e2)
        i2
   case (e3)
        i3
   case (e4)
        ...
   otherwise
        in
```

sfobs

**while.** *Sintaxa*:

```
while ⟨expresie⟩ do
   ⟨secvenţă-instrucţiuni⟩
```

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean, iar ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.

*Semantica*: 1. Se evaluează ⟨expresie⟩.

2. Dacă rezultatul evaluării este `true`, atunci se execută ⟨secvenţă-instrucţiuni⟩, după care se reia procesul începând cu pasul 1. Dacă rezultatul evaluării este `false`, atunci execuţia instrucţiunii `while` se termină.

**for.** *Sintaxa*:

```
for ⟨variabilă⟩ ← ⟨expresie1⟩ to ⟨expresie2⟩ do
   ⟨secvenţă-instrucţiuni⟩
```

sau

```
for ⟨variabilă⟩ ← ⟨expresie1⟩ downto ⟨expresie2⟩ do
   ⟨secvenţă-instrucţiuni⟩
```

unde ⟨variabilă⟩ este o variabilă de tip întreg, ⟨expresie$_i$⟩, $i = 1, 2$, sunt expresii care prin evaluare dau valori întregi, ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.

*Semantica*: Instrucţiunea

```
for i ← e1 to e2 do
   S
```

simulează execuţia următorului program:

```
i ← e1
temp ← e2
while (i ≤ temp) do
   S
   i ← succ(i)
   /* succ(i) întoarce succesorul numărului întreg i */
```

iar instrucţiunea:

```
for i ← e1 downto e2 do
    S
```

simulează execuţia următorului program:

```
i ← e1
temp ← e2
while (i ≥ temp) do
    S
    i ← pred(i)
    /* pred(i) intoarce predecesorul numarului intreg i */
```

O formă mai puţin convenţională a instrucţiunii `for`, dar frecvent utilizată în programe este `for each`:

**for each.** *Sintaxa*:

```
for each ⟨variabilă⟩ ∈ ⟨mulţime⟩ do
    ⟨secvenţă-instrucţiuni⟩
```

unde ⟨variabilă⟩ este o variabilă de tip $\mathcal{T}$, ⟨mulţime⟩ este o mulţime finită de valori din $\mathcal{T}$, ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.

*Semantica*: Instrucţiunea

```
for each x ∈ T do
    S
```

simulează execuţia următorului program:

```
while (T ≠ ∅) do
    x ← un element din T
    S
    T ← T\{x}
```

**repeat.** *Sintaxa*:

```
repeat
    ⟨secvenţă-instrucţiuni⟩
until ⟨expresie⟩
```

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean, iar ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.

*Semantica*: Instrucţiunea

```
repeat
    S
until e
```

simulează execuţia următorului program:

```
S
while (not e) do
    S
```

**Excepţii.** *Sintaxa*:

```
throw ⟨mesaj⟩
```

unde ⟨mesaj⟩ este un şir de caractere (un text).
*Semantica*: Execuţia programului se opreşte şi este afişat textul ⟨mesaj⟩. Cel mai adesea, `throw` este utilizată împreună cu `if`:

```
if ⟨expresie⟩ then throw ⟨mesaj⟩
```

Obţinerea rezultatului `true` în urma evaluării expresiei are ca semnificaţie apariţia unei excepţii, caz în care execuţia programului se opreşte. Un exemplu de excepţie este cel când procedura `new` nu poate aloca memorie pentru variabilele dinamice:

```
new(p)
if (p = NULL) then throw 'memorie insuficienta'
```

### 1.1.5 Subprograme

Limbajul nostru algoritmic este unul modular, unde un modul este identificat de un subprogram. Există două tipuri de subprograme: proceduri şi funcţii.

**Proceduri.** Interfaţa dintre o procedură şi modulul care o apelează este realizată numai prin parametri şi variabilele globale. De aceea, apelul unei proceduri apare ca o instrucţiune separată. Forma generală a unei proceduri este:

```
procedure ⟨nume⟩(⟨lista-parametri⟩)
   ⟨secvenţă-instrucţiuni⟩
end
```

Lista parametrilor este opţională. Considerăm ca exemplu o procedură care interschimbă valorile a două variabile:

```
procedure swap(x, y)
   aux ← x
   x ← y
   y ← aux
end
```

Permutarea circulară a valorilor a trei variabile `a, b, c` se face apelând de două ori procedura `swap`:

```
swap(a, b)
swap(b, c)
```

**Funcţii.** În plus faţă de proceduri, o funcţie întoarce o valoare asociată cu numele funcţiei. De aceea, apelul unei funcţii poate participa la formarea de expresii. Forma generală a unei funcţii este:

```
function ⟨nume⟩(⟨lista-parametri⟩)
   ⟨secvenţă-instrucţiuni⟩
   return ⟨expresie⟩
end
```

Lista parametrilor este opţională. Valoarea întoarsă de funcţie este cea obţinută prin evaluarea expresiei. O instrucţiune `return` poate apărea în mai multe locuri în definiţia unei funcţii şi execuţia ei implică terminarea evaluării funcţiei. Considerăm ca exemplu o funcţie care calculează maximul dintre valorile a trei variabile:

```
function max3(x, y, z)
   temp ← x
   if (y > temp) then temp ← y
   if (z > temp) then temp ← z
   return temp
end
```

Are sens să scriem `2*max3(a, b, c)` sau `max3(a, b, c) < 5`.

#### 1.1.5.1 Comentarii

Comentariile sunt notate în mod similar cu limbajul C, utilizând combinaţiile de caractere `/*` şi `*/`. Comentariile au rolul de a introduce explicaţii suplimentare privind descrierea algoritmului:

```
procedure absDec(x)
   if (x > 0) /* testeaza daca x este pozitiv */
      then x ← x-1 /* decrementeaza x */
      else x ← x+1 /* incrementeaza x */
end
```

### 1.1.6 Tipuri de date structurate de nivel jos

#### 1.1.6.1 Tablouri

Un *tablou* este un ansamblu omogen de variabile, numite *componentele* tabloului, în care toate variabilele componente aparţin aceluiaşi tip şi sunt identificate cu ajutorul indicilor. Un *tablou 1-dimensional* (*unidimensional*) este un tablou în care componentele sunt identificate cu ajutorul unui singur indice. De exemplu, dacă numele variabilei tablou este `a` şi mulţimea valorilor pentru indice este $\{0, 1, 2, \ldots, n-1\}$, atunci variabilele componente sunt `a[0]`, `a[1]`, `a[2]`, ..., `a[n-1]`. Memoria alocată unui tablou 1-dimensional este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare componentă. Ordinea de memorare a componentelor este dată de ordinea indicilor. Tablourile 1-dimensionale sunt reprezentate grafic ca în figura 1.5. Operaţiile asupra tablourilor se realizează prin intermediul componentelor. Prezentăm ca exemplu iniţializarea tuturor componentelor cu 0:

```
for i ← 0 to n-1 do
  a[i] ← 0
```

Un *tablou 2-dimensional* (*bidimensional*) este un tablou în care componentele sunt identificate cu ajutorul a doi indici. De exemplu, dacă numele variabilei tablou este `a` şi mulţimea valorilor pentru primul indice este $\{0, 1, \ldots, m-1\}$, iar mulţimea valorilor pentru cel de-al doilea indice este $\{0, 1, \ldots, n-1\}$ atunci variabilele componente sunt `a[0,0]`, `a[0,1]`, ..., `a[0,n-1]`, ..., `a[m-1,0]`, `a[m-1,1]`, ..., `a[m-1,n-1]`. Ca şi în cazul tablourilor 1-dimensionale, memoria alocată unui tablou

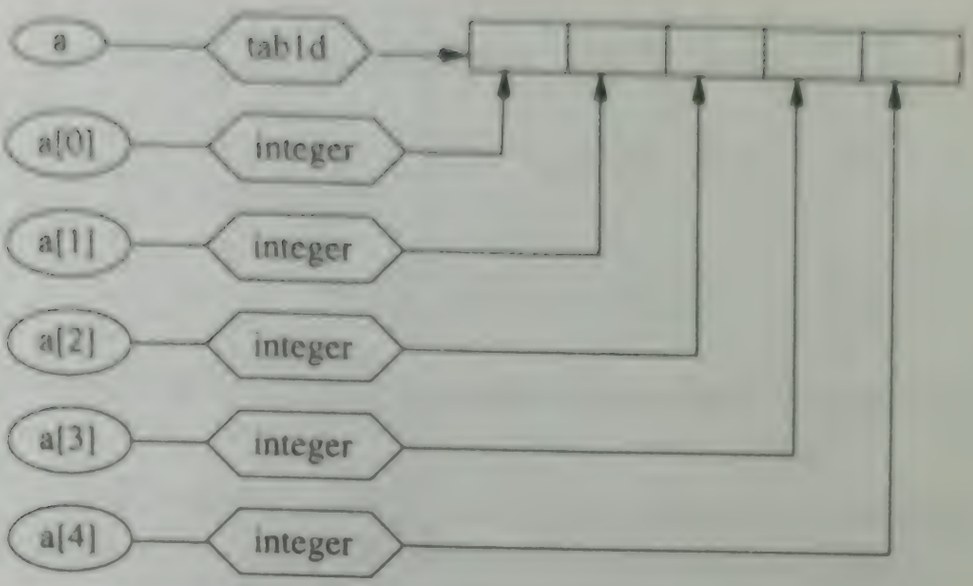



Figura 1.5: Tablou 1-dimensional

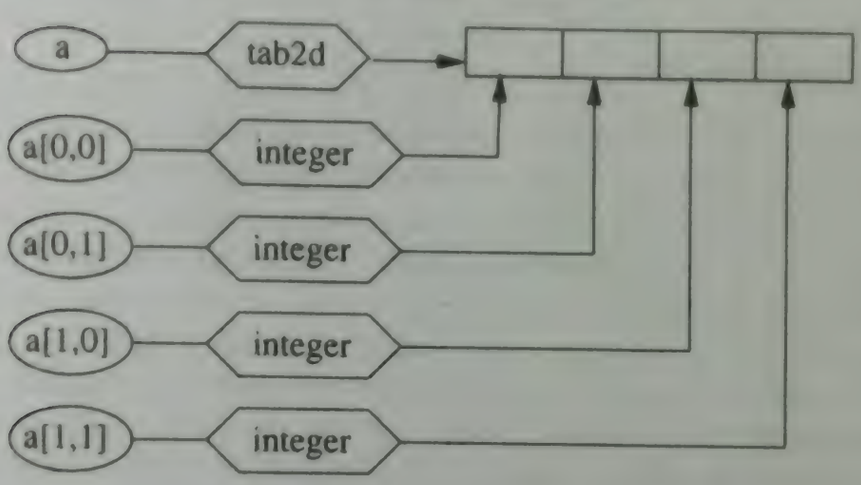



Figura 1.6: Tablou 2-dimensional

2-dimensional este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare componentă. Ordinea de memorare a componentelor este dată de ordinea lexicografică definită peste indici ($[i,j] < [k,l]$ ddacă $i < k$ sau $i = k$ şi $j < l$). Tablourile 2-dimensionale sunt reprezentate grafic ca în figura 1.6. Aşa cum o matrice poate fi văzută ca un vector de linii, la fel un tablou 2-dimensional poate fi văzut ca un tablou 1-dimensional de tablouri 1-dimensionale. Din acest motiv, componentele unui tablou 2-dimensional mai pot fi notate prin expresii de forma `a[0][0]`, `a[0][1]` etc. (a se vedea şi figura 1.7).

#### 1.1.6.2 Şiruri de caractere

Şirurile de caractere sunt secvenţe de caractere. Pot fi asociate cu tablourile unidimensionale ale căror elemente sunt caractere. O constantă şir de caractere este notată utilizând convenţia din limbajul C: `'exemplu de sir'`. Peste şiruri sunt definite următoarele operaţii:

- *concatenarea*, notată cu +: `'unu'+'doi'` are ca rezultat `'unudoi'`;

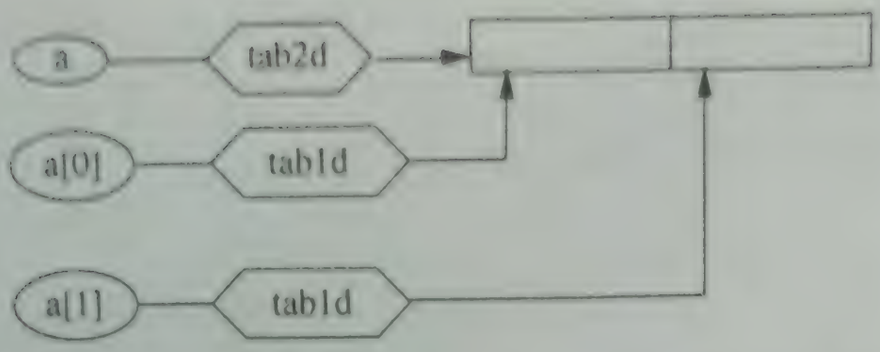



Figura 1.7: Tablou 2-dimensional văzut ca tablou de tablouri 1-dimensionale

- `strcmp(sir1, sir2)` întoarce rezultatul comparării lexicografice a celor două şiruri: −1 dacă `sir1` < `sir2`, 0 dacă `sir1` = `sir2`, şi +1 dacă `sir1` > `sir2`;
- `strlen(sir)` întoarce lungimea şirului dat ca parametru;
- `strcpy(sir1, sir2)` realizează copierea şirului `sir2` în `sir1`.

#### 1.1.6.3 Structuri statice

O *structură* statică este un ansamblu eterogen de variabile, numite *câmpuri*, în care fiecare câmp are propriul său nume şi propriul său tip. Numele complet al unui câmp se obţine din numele structurii urmat de caracterul '.' şi numele câmpului. Memoria alocată unei structuri statice este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare câmp. Componenţa unei structuri statice nu se schimbă în timp, adică nu pot fi adaugate sau şterse câmpuri. Ordinea de memorare a câmpurilor corespunde cu ordinea de descriere a acestora în cadrul structurii. Ca exemplu, presupunem că o figură `f` este descrisă de două câmpuri: `f.pozitie` – punctul care precizează poziţia figurii, şi `f.vizibil` – valoare booleana care precizează dacă figura este desenată sau nu. La rândul său, punctul poate fi văzut ca o structură statică cu două câmpuri – câte unul pentru fiecare coordonată (considerăm numai puncte în plan având coordonate întregi). Structurile statice sunt reprezentate grafic ca în figura 1.8. Pentru identificarea câmpurilor unei structuri statice referite indirect prin intermediul unui pointer, vom utiliza o notaţie similară celei din limbajul C (a se vedea şi figura 1.9).

#### 1.1.6.4 Structuri dinamice înlănţuite

O *structură dinamică înlănţuită* este o înlănţuire de structuri statice, numite *noduri*, în care pot fi inserate sau eliminate noduri, cu condiţia să fie păstrată coerenţa înlănţuirii.

O *structură dinamică simplu înlănţuită* este o înlănţuire de structuri statice (noduri), în care fiecare nod „cunoaşte" adresa nodului de după el (nodul succesor). Poate face excepţie de la această regulă ultimul nod. În forma sa cea mai simplă, un nod `v` este o structură statică cu două câmpuri: un câmp `v->elt` pentru memorarea informaţiei şi un câmp `v->succ` care memorează adresa nodului succesor. Se presupune că se cunosc adresele primului şi respectiv ultimului nod din structură. O structură simplu înlănţuită este reprezentată grafic ca în figura 1.10. Structura

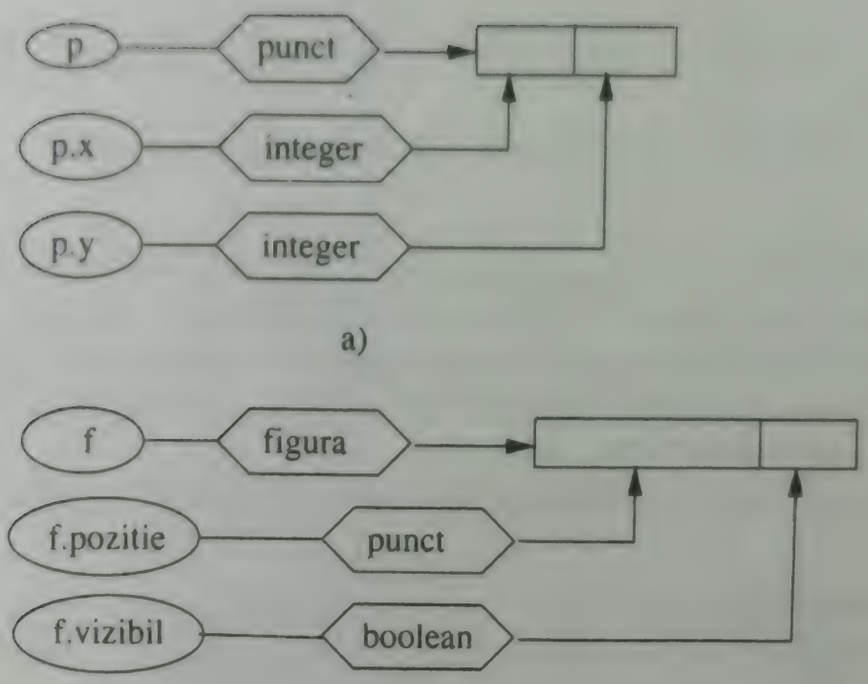



Figura 1.8: Structuri statice

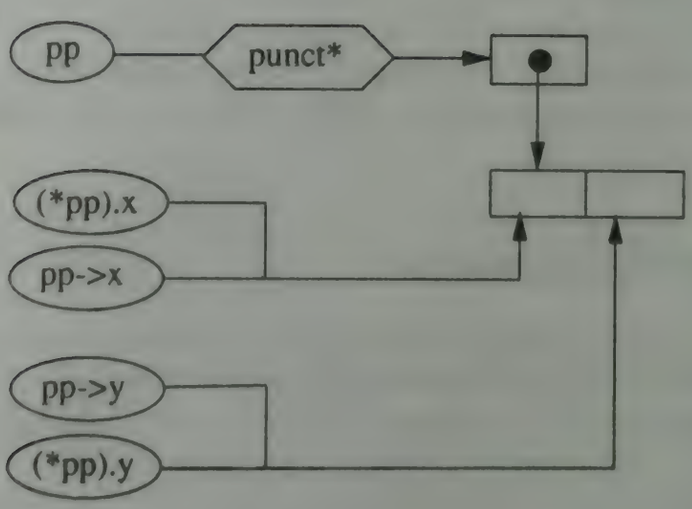



Figura 1.9: Structuri şi pointeri

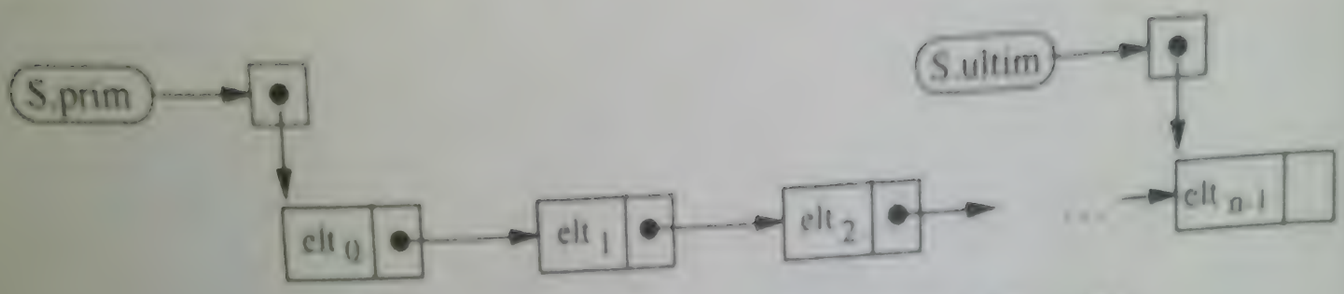



Figura 1.10: Structură dinamică simplu înlănţuită

simplu înlănţuită este o structură de date dinamică, în sensul că pot fi inserate sau eliminate noduri, cu condiţia să fie păstrată proprietatea de înlănţuire liniară.

Operaţiile de bază ce se pot efectua asupra unei structuri simplu înlănţuite sunt adăugarea sau ştergerea unui nod la începutul, în interiorul sau la sfârşitul structurii, parcurgerea structurii.

Prezentăm ca exemplu adăugarea unui nod la începutul structurii (a se vedea figura 1.11):

```
procedure adLaInc(S, e)
   new(v)
   v->elt ← e
   if (S.prim = NULL)
      then S.prim ← v          /* structura vida */
           S.ultim ← v
           v->succ ← NULL
      else v->succ ← S.prim    /* structura nevida */
           S.prim ← v
end
```

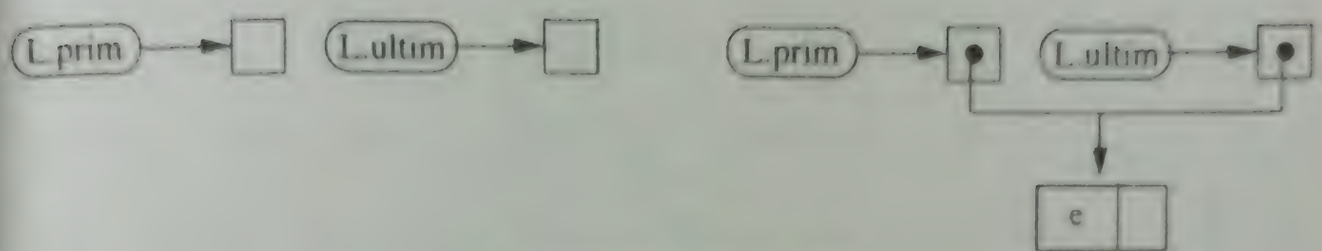
*Structurile dinamice dublu înlănţuite* sunt asemănătoare celor simplu înlănţuite cu deosebirea că, în plus, fiecare nod „cunoaşte" adresa nodului din faţa sa (nodul predecesor). Astfel, un nod v are un câmp în plus, v->pred, care memorează adresa nodului predecesor. Primul nod poate face excepţie de la regula „cunoaşterii" predecesorului. O structură dinamică dublu înlănţuită este reprezentată grafic ca în figura 1.12.

### 1.1.7 Execuţia programelor

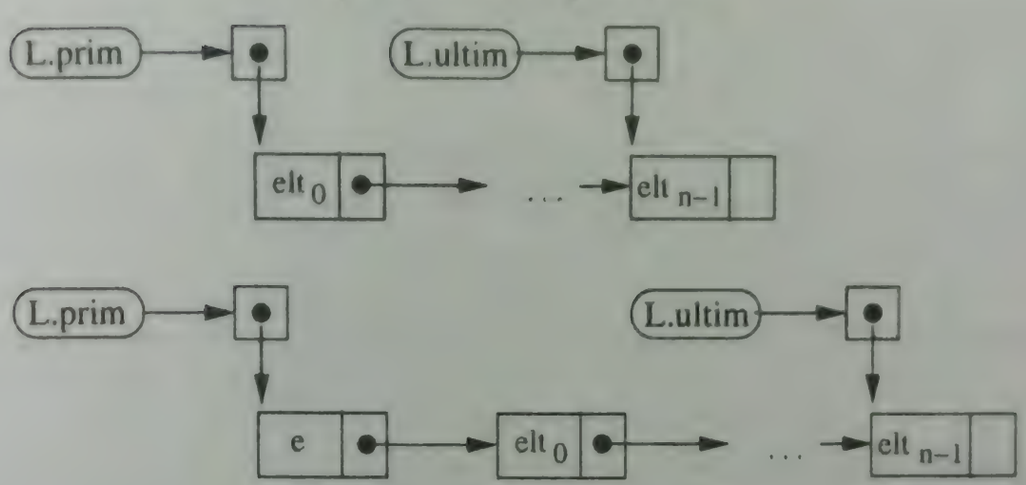
Intuitiv, o execuţie a unui program constă in succesiunea de paşi elementari determinaţi de execuţiile instrucţiunilor ce compun programul. Convenim să utilizăm noţiunea de *calcul* pentru execuţia unui program. Fie, de exemplu, următorul program:

```
x ← 0
i ← 1
while (i < 10) do
   x ← x*10+i
   i ← i+2
```

Calculul descris de acest program ar putea fi reprezentat de următorul tabel:




a) Structura iniţială vidă




b) Structura iniţială nevidă

Figura 1.11:
Adăugarea unui nod la începutul unei structuri dinamice simplu înlănţuite

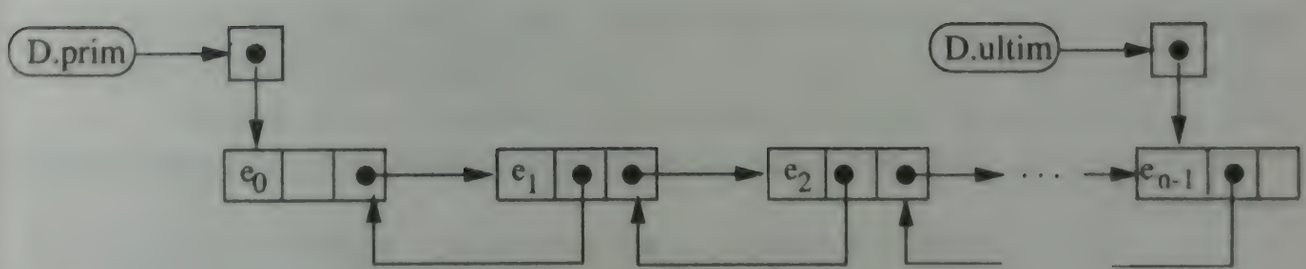



Figura 1.12: Structură dinamică dublu înlănţuită

| Pasul | Instrucţiunea | i | x |
|---|---|---|---|
| 0 | x ← 0 | – | – |
| 1 | i ← 1 | – | 0 |
| 2 | 1<10 | 1 | 0 |
| 3 | x ← x*10+i | 1 | 0 |
| 4 | i ← i+2 | 1 | 1 |
| 5 | 3<10 | 3 | 1 |
| 6 | x ← x*10+i | 3 | 1 |
| 7 | i ← i+2 | 3 | 13 |
| 8 | 5<10 | 5 | 13 |
| 9 | x ← x*10+i | 5 | 13 |
| 10 | i ← i+2 | 5 | 135 |
| 11 | 7<10 | 7 | 135 |
| 12 | x ← x*10+i | 7 | 135 |
| 13 | i ← i+2 | 7 | 1357 |
| 14 | 9<10 | 9 | 1357 |
| 15 | x ← x*10+i | 9 | 1357 |
| 16 | i ← i+2 | 9 | 13579 |
| 17 | 11<10 | 11 | 13579 |
| 18 | | 11 | 13579 |

Acest calcul este notat formal printr-o secvenţă $c_0 \vdash c_1 \vdash \cdots \vdash c_{18}$. Prin $c_i$ am notat configuraţiile ce intervin în calcul. O *configuraţie* include instrucţiunea curentă (starea programului) şi starea memoriei (valorile curente ale variabilelor din program). În exemplul de mai sus, o configuraţie este reprezentată de o linie în tabel. Relaţia $c_{i-1} \vdash c_i$ are următoarea semnificaţie: prin execuţia instrucţiunii din $c_{i-1}$, se transformă $c_{i-1}$ în $c_i$. $c_0$ se numeşte *configuraţie iniţială*, iar $c_{18}$ *configuraţie finală*. Notăm că pot exista şi calcule infinite. De exemplu, instrucţiunea:

```
while (true) do
    i ← i+1
```

generează un calcul infinit.

### 1.1.8 Exerciţii

**Exerciţiul 1.1.1.** O *secţiune* a tabloului a, notată a[i..j], este formată din elementele a[i], a[i + 1], . . . , a[j], $i \leq j$. *Suma* unei secţiuni este suma elementelor sale. Să se scrie un program care determină secţiunea de sumă maximă.

**Exerciţiul 1.1.2.** Să se scrie o funcţie care, pentru un tablou b, determină valoarea predicatului:

$$P \equiv (\forall i, j) 10 \leq i < j \leq n-1 \Rightarrow b[i] \leq b[j]$$

Să se modifice acest subprogram astfel încât să ordoneze crescător elementele unui tablou.

**Exerciţiul 1.1.3.** Să se scrie un subprogram de tip funcţie care, pentru un tablou de valori booleene b, determină valoarea predicatului:

$$P \equiv (\forall i, j) 0 \leq i \leq j \leq n-1 \Rightarrow (b[i] \Rightarrow b[j]))$$

**Exerciţiul 1.1.4.** Se consideră tabloul a ordonat crescător şi tabloul b ordonat descrescător. Să se scrie un program care determină cel mai mic $x$, când există, ce apare în ambele tablouri.

**Exerciţiul 1.1.5.** Să se scrie un program care să pună elementele unei matrici pătratice într-o listă simplu înlănţuită, în ordinea parcurgerii acesteia în spirală, începând cu colţul stânga-jos. De exemplu, pentru matricea $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ ordinea parcurgerii este $a_{31}, a_{32}, a_{33}, a_{23}, a_{13}, a_{12}, a_{11}, a_{21}, a_{22}$. Matricea va fi reprezentată ca tablou 2-dimensional.

**Exerciţiul 1.1.6.** Să se proiecteze o structură statică pentru reprezentarea numerelor complexe. Să se scrie subprograme care realizează operaţii din algebra numerelor complexe.

**Exerciţiul 1.1.7.** Presupunem că numerele complexe sunt reprezentate ca în soluţia exerciţiului 1.1.6. Un polinom cu coeficienţi complecşi poate fi reprezentat ca un tablou de structuri statice. Să se scrie un subprogram care, pentru un polinom cu coeficienţi complecşi $P$ şi un număr complex $z$ date, calculează $P(z)$.

**Exerciţiul 1.1.8.** Să se proiecteze o structură statică pentru reprezentarea datei calendaristice. Să se scrie subprograme care realizează următoarele:

a) Decide dacă valoarea unei variabile conţine o dată validă.
b) Având la intrare o dată calendaristică oferă la ieşire data calendaristică următoare.
c) Având la intrare o dată calendaristică oferă la ieşire data calendaristică precedentă.

**Exerciţiul 1.1.9.** Dat fiind un şir de numere întregi, să se scrie subprograme pentru următoarele operaţii:

a) Plasarea elementelor şirului într-o structură liniară simplu înlănţuită, prin inserări repetate în faţa structurii.
b) Parcurgerea structurii create.

**Exerciţiul 1.1.10.** Submulţimile finite de numere naturale $\mathcal{M} \subset \mathcal{Q}$ pot fi reprezentate prin tablouri. Să se scrie subprograme pentru operaţiile $\in$, $\subset$, $\cup$, $\cap$, $\setminus$.

**Exerciţiul 1.1.11.** Să se definească o implementare cu structuri dinamice simplu înlănţuite, pentru mulţimile $\mathcal{M}$ din exerciţiul anterior şi să se scrie subprograme pentru operaţiile specifice mulţimilor.

## 1.2 Probleme, algoritmi şi programe

Noţiunile de algoritmi, programe şi probleme sunt diferite şi trebuie înţeles bine care sunt relaţiile dintre ele. Un algoritm exprimă o modalitate de a obţine soluţia unei probleme specifice. Descrierea algoritmului într-un limbaj algoritmic se face prin intermediul unui program. Toate noţiunile definite pentru program sunt extinse în mod natural la algoritm. Astfel, putem vorbi despre execuţia (calculul) unui algoritm, configuraţie iniţială, etc. În această secţiune vom preciza ce înseamnă că un algoritm exprimă o modalitate de a obţine soluţia pentru o problemă dată.

### 1.2.1 Noţiunea de problemă

O problemă are două componente: domeniul, care descrie elementele ce intervin în problemă şi relaţiile dintre aceste elemente şi o întrebare despre proprietăţile anumitor elemente sau o cerinţă de determinare a unor elemente ce au o anumită proprietate. În funcţie de scopul urmărit, există mai multe moduri de a formaliza o problemă. Noi vom utiliza numai două dintre ele.

**Intrare/Ieşire.** Putem formaliza problema rezolvată de algoritm ca o pereche (*Intrare*, *Ieşire*). Componenta *Intrare* descrie datele de intrare, iar componenta *Ieşire* descrie datele de ieşire. Un exemplu simplu de problemă reprezentată astfel este următorul:

*Intrare:* Un număr întreg pozitiv $x$.
*Ieşire:* Cel mai mare număr prim mai mic decât sau egal cu $x$.

**Problemă de decizie.** Este un caz particular de problemă când ieşirea este de forma *'DA'* sau *'NU'*. O astfel de problemă este reprezentată ca o pereche (*Instanţă*, *Întrebare*) unde componenta *Instanţă* descrie datele de intrare, iar componenta *Întrebare* se referă, în general, la existenţa unui obiect sau a unei proprietăţi. Un exemplu tipic îl reprezintă următoarea problemă:

*Instanţă:* Un număr întreg pozitiv $x$.
*Întrebare:* Este $x$ număr prim?

Problemele de decizie sunt preferate atât în teoria complexităţii, cât şi în teoria calculabilităţii datorită reprezentării foarte simple a ieşirilor. Facem observaţia că orice problemă admite o reprezentare sub formă de problemă de decizie, indiferent de reprezentarea sa iniţială. Un exemplu de reprezentare a unei probleme de optim ca problemă de decizie este dat în secţiunea 13.1.

De obicei, pentru reprezentarea problemelor de decizie se consideră o mulţime $\mathcal{U}$, iar o instanţă este de forma $\mathcal{R} \subseteq \mathcal{U}, x \in \mathcal{U}$ şi întrebarea de forma $x \in \mathcal{R}$?. Problema din exemplul anterior poate fi reprezentată astfel:

*Instanţă:* $\mathcal{P}$ = mulţimea numerelor prime ( $\mathcal{P} \subset \mathbb{Z}$), $x \in \mathbb{Z}^+$.
*Întrebare:* $x \in \mathcal{P}$?

### 1.2.2 Problemă rezolvată de un algoritm

Convenim să considerăm întotdeauna intrările $p$ ale unei probleme $P$ ca fiind instanţe şi, prin abuz de notaţie, scriem $p \in P$.

**Definiţia 1.1.** *Fie $A$ un algoritm şi $P$ o problemă. Spunem că o configuraţie iniţială $c_0$ a lui $A$* include *instanţa $p \in P$ dacă există o structură de date* `inp`*, definită în $A$, astfel încât valoarea lui* `inp` *din $c_0$ constituie o reprezentare a lui $p$. Analog, spunem că o configuraţie finală $c_n$ a unui algoritm $A$* include *ieşirea $P(p)$ dacă există o structură de date* `out`*, definită în $A$, astfel încât valoarea lui* `out` *din $c_n$ constituie o reprezentare a lui $P(p)$.*

**Definiţia 1.2.** *(Problemă rezolvată de un algoritm)*
1. Un algoritm $A$ *rezolvă o problemă* $P$ *în sensul corectitudinii totale* dacă pentru orice instanţă $p$, calculul unic determinat de configuraţia iniţială ce include $p$ este finit şi configuraţia finală include ieşirea $P(p)$.
2. Un algoritm $A$ *rezolvă o problemă* $P$ *în sensul corectitudinii parţiale* dacă pentru orice instanţă $p$ pentru care calculul unic determinat de configuraţia iniţială ce include $p$ este finit, configuraţia finală include ieşirea $P(p)$.

Ori de câte ori spunem că un algoritm $A$ rezolvă o problemă $P$ vom înţelege de fapt că $A$ rezolvă o problemă $P$ în sensul corectitudinii totale.

**Definiţia 1.3.** *(Problemă rezolvabilă/nerezolvabilă)*
1. O problemă $P$ este *rezolvabilă* dacă există un algoritm care să o rezolve în sensul corectitudinii totale. Dacă $P$ este o problemă de decizie, atunci spunem că $P$ este *decidabilă*.
2. O problemă de decizie $P$ este *semidecidabilă* sau *parţial decidabilă* dacă există un algoritm $A$ care rezolvă $P$ în sensul corectitudinii parţiale astfel încât calculul lui $A$ este finit pentru orice instanţă $p$ pentru care răspunsul la întrebare este *'DA'*.
3. O problemă $P$ este *nerezolvabilă* dacă nu este rezolvabilă, adică nu există un algoritm care să o rezolve în sensul corectitudinii totale. Dacă $P$ este o problemă de decizie, atunci spunem că $P$ este *nedecidabilă*.

### 1.2.3 Un exemplu de problemă nedecidabilă

Pentru a arăta că o problemă este decidabilă este suficient să găsim un algoritm care să o rezolve. Mai complicat este cazul problemelor nedecidabile. În legătură cu acestea din urmă se pun, în mod firesc, următoarele întrebări: Există probleme nedecidabile? Cum putem demonstra că o problemă nu este decidabilă? Răspunsul la prima întrebare este afirmativ. Un prim exemplu de problemă nedicidabilă este cea cunoscută sub numele de *problema opririi*. Notăm cu $\mathcal{U}$ mulţimea perechilor de forma $\langle A, \overline{x} \rangle$ unde $A$ este un algoritm şi $\overline{x}$ este o intrare pentru $A$, iar $\mathcal{R}$ este submulţimea formată din acele perechi $\langle A, \overline{x} \rangle$ pentru care calculul lui $A$ pentru intrarea $\overline{x}$ este finit. Dacă notăm prin $A(\overline{x})$ (a se citi $A(\overline{x}) = true$) faptul că $\langle A, \overline{x} \rangle \in \mathcal{R}$, atunci problema opririi poate fi scrisă astfel:

**Problema opririi**
*Instanţă:* Un algoritm $A$, $\overline{x} \in \mathbb{Z}^*$.
*Întrebare:* $A(\overline{x})$?

**Teorema 1.1.** *Problema opririi nu este decidabilă.*

*Demonstraţie.* Un algoritm $AO$, care rezolvă problema opririi, are ca intrare o pereche $\langle A, \overline{x} \rangle$ şi se opreşte întotdeauna cu răspunsul *'DA'*, dacă $A$ se opreşte pentru intrarea $\overline{x}$, sau cu răspunsul *'NU'*, dacă $A$ nu se opreşte pentru intrarea $\overline{x}$. Fie $T$ următorul algoritm:

```
if (AO(x̄,x̄) = 'DA')
   then while (true) do /*nimic*/
```

Algoritmul $T$ nu se opreşte (bucla `while` se execută la infinit), dacă $AO(\overline{T}, \overline{T})$ întoarce 'DA'. Reamintim că $AO(\overline{T}, \overline{T}) = 'DA'$ înseamnă că algoritmul reprezentat de $\overline{T}$ se opreşte pentru intrarea $\overline{T}$, adică propria sa codificare. Presupunem acum că $\overline{T}$ este codificarea lui $T$. Există două posibilităţi.

1. $T$ se opreşte pentru intrarea $\overline{T}$. Rezultă $AO(\overline{T}, \overline{T}) = 'NU'$, adică algoritmul reprezentat de $\overline{T}$ nu se opreşte pentru intrarea $\overline{T}$. Dar algoritmul reprezentat de $\overline{T}$ este chiar $T$. Contradicţie.
2. $T$ nu se opreşte pentru intrarea $\overline{T}$. Rezultă $AO(\overline{T}, \overline{T}) = 'DA'$, adică algoritmul reprezentat de $\overline{T}$ se opreşte pentru intrarea $\overline{T}$. Contradicţie.

Aşadar, nu există un algoritm $AO$ care să rezolve problema opririi. sfdem

**Observaţie.** Rezolvarea teoremei de mai sus este strâns legată de următorul paradox logic. „Există un oraş cu un bărbier care bărbiereşte pe oricine ce nu se bărbiereşte singur. Cine îl bărbiereşte pe bărbier?" sfobs

## 1.3 Măsurarea performanţelor unui algoritm

Evaluarea algoritmilor din punctul de vedere al performanţelor obţinute de aceştia în rezolvarea problemelor este o etapă esenţială în procesul de decizie a utilizării acestora în aplicaţii. La evaluarea (estimarea) algoritmilor se pune în evidenţă consumul celor două resurse fundamentale: timpul de execuţie şi spaţiul de memorare a datelor. În funcţie de priorităţile alese, se aleg limite pentru resursele timp şi spaţiu. Algoritmul este considerat eligibil dacă consumul celor două resurse se încadrează în limitele stabilite.

Fie $P$ o problemă şi $A$ un algoritm pentru $P$. Fie $c_0 \vdash_A c_1 \cdots \vdash_A c_n$ un calcul finit al algoritmului $A$. Notăm cu $t_A(c_i)$ timpul necesar obţinerii configuraţiei $c_i$ din $c_{i-1}$, $1 \leq i \leq n$, şi cu $s_A(c_i)$ spaţiul de memorie ocupat în configuraţia $c_i$, $0 \leq i \leq n$.

**Definiţia 1.4.** *Fie $A$ un algoritm pentru problema $P$, $p \in P$ o instanţă a problemei $P$ şi $c_0 \vdash c_1 \vdash \cdots \vdash c_n$ calculul lui $A$ corespunzător instanţei $p$.* Timpul *necesar algoritmului $A$ pentru rezolvarea instanţei $p$ este:*

$$T_A(p) = \sum_{i=1}^{n} t_A(c_i)$$

Spaţiul *(de memorie) necesar algoritmului $A$ pentru rezolvarea instanţei $p$ este:*

$$S_A(p) = \max_{0 \leq i \leq n} s_A(c_i)$$

În general sunt dificil de calculat cele două valori în funcţie de instanţe. Această dificultate poate fi simplificată astfel. Asociem unei instanţe $p \in P$ o mărime $g(p)$, care este un număr natural, pe care o numim *mărimea* instanţei $p$. De exemplu, $g(p)$ poate fi suma lungimilor reprezentărilor corespunzând datelor din instanţa $p$. Dacă reprezentările datelor din $p$ au aceeaşi lungime, atunci se poate considera $g(p)$

egală cu numărul datelor. Astfel, dacă $p$ constă dintr-un tablou atunci se poate lua $g(p)$ ca fiind numărul de elemente ale tabloului; dacă $p$ constă dintr-un polinom se poate considera $g(p)$ ca fiind gradul polinomului (= numărul coeficienţilor minus 1); dacă $p$ este un graf se poate lua $g(p)$ ca fiind numărul de vârfuri, numărul de muchii sau numărul de vârfuri + numărul de muchii etc.

**Definiţia 1.5.** *Fie $A$ un algoritm pentru problema $P$.*

1. *Spunem că $A$ rezolvă $P$ în* timpul $T_A^{fav}(n)$ *dacă:*

$$T_A^{fav}(n) = \inf\{T_A(p) \mid p \in P, g(p) = n\}$$

2. *Spunem că $A$ rezolvă $P$ în* timpul $T_A(n)$ *dacă:*

$$T_A(n) = \sup\{T_A(p) \mid p \in P, g(p) = n\}$$

3. *Spunem că $A$ rezolvă $P$ în spaţiul $S_A^{fav}(n)$ dacă:*

$$S_A^{fav}(n) = \inf\{S_A(p) \mid p \in P, g(p) = n\}$$

4. *Spunem că $A$ rezolvă $P$ în spaţiul $S_A(n)$ dacă:*

$$S_A(n) = \sup\{S_A(p) \mid p \in P, g(p) = n\}$$

*Funcţia $T_A^{fav}$ ($S_A^{fav}$)* se numeşte *timpul de execuţie al algoritmului (spaţiul utilizat de algoritmul) $A$ pentru cazul cel mai favorabil*, iar funcţia $T_A$ ($S_A$) se numeşte *timpul de execuţie al algoritmului (spaţiu utilizat de algoritmul) $A$ pentru cazul cel mai nefavorabil.*

**Exemplu.**

Considerăm problema căutării unui element într-o secvenţă de numere întregi.

**Problema** $P_1$

*Intrare:* $n, (a_0, \ldots, a_{n-1}), z$ numere întregi.

*Ieşire:* $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

Presupunem că secvenţa $(a_0, \ldots, a_{n-1})$ este memorată în tabloul $(\mathtt{a[i]} \mid 0 \leq i \leq n-1)$. Algoritmul $A_1$ descris de următorul program rezolvă $P_1$:

```
/* algoritmul A1 */
i ← 0
while (a[i] ≠ z) and (i < n-1) do
   i ← i+1
if (a[i] = z)
   then poz ← i
   else poz ← -1
```

Considerăm ca dimensiune a problemei $P_1$ numărul $n$ al elementelor din secvenţa în care se caută. Deoarece suntem în cazul când toate datele sunt memorate pe câte un cuvânt de memorie, vom presupune că toate operaţiile necesită o unitate de timp. Calculăm timpii de execuţie. Cazul cel mai favorabil este obţinut când $a_0 = z$ şi se

efectuează trei comparaţii şi două atribuiri. Rezultă $T_{A_1}^{fav}(n) = 3 + 2 = 5$. Cazul cel mai nefavorabil se obţine când $x \notin \{a_0, \ldots, a_{n-1}\}$ sau $x = a[n-1]$, în acest caz fiind executate $2n + 1$ comparaţii şi $1 + (n-1) + 1 = n + 1$ atribuiri. Rezultă $T_{A_1}(n) = 3n + 2$. Pentru simplitatea prezentării, nu au mai fost luate în considerare operaţiile and şi operaţiile de adunare şi scădere. Spaţiul utilizat de algoritm, pentru ambele cazuri, este $n + 7$ (tabloul a, constantele 0, 1 şi -1, variabilele i, poz, n şi z). sfex

**Exemplu.** Considerăm acum problema determinării elementului de valoare maximă dintr-o secvenţă de numere întregi:

**Problema $P_2$**

*Intrare:* $n, (a_0, \ldots, a_{n-1})$ numere întregi.

*Ieşire:* $max = \max\{a_i \mid 0 \le i \le n-1\}$.

Şi aici presupunem că secvenţa $(a_0, \ldots, a_{n-1})$ este memorată în tabloul $(\mathtt{a[i]} \mid 0 \le i \le n-1)$. Algoritmul $A_2$ descris de următorul program rezolvă $P_2$:

```
/* algoritmul A2 */
max ← a[0]
for i ← 1 to n-1 do
   if (a[i]>max)
      then max ← a[i]
```

Dimensiunea problemei $P_2$ este $n$, numărul elementelor din secvenţa în care se caută maximul. Vom presupune şi în acest caz că toate operaţiile necesită o unitate de timp. Cazul cel mai favorabil este obţinut când $a_0$ este elementul de valoare maximă. Se efectuează $n + n - 1 = 2n - 1$ comparaţii şi $1 + n$ atribuiri. Rezultă $T_{A_2}^{fav}(n) = 2n - 1 + 1 + n = 3n$. Cazul cel mai defavorabil se obţine în situaţia în care tabloul este ordonat crescător (pentru că de fiecare dată instrucţiunea `if` se execută pe ramura `then`, adică se face şi comparaţie şi atribuire). În acest caz numărul comparaţiilor este $2n - 1$ şi cel al atribuirilor este $2n$. Rezultă $T_{A_2}(n) = 2n - 1 + 2n = 4n - 1$. Dimensiunea memoriei utilizate de algoritm este $n+5$ (tabloul a, constanta 1, variabilele i, n şi z), în ambele cazuri. sfex

**Exemplu.** Sortarea prin inserare.

**Problema $P_3$**

*Intrare:* $n, (a_0, \ldots, a_{n-1})$ numere întregi.

*Ieşire:* $(a_{i_0}, \ldots, a_{i_{n-1}})$ unde $(i_0, \ldots, i_{n-1})$ este o permutare a şirului $(0, \ldots, n-1)$ şi $a_{i_j} \le a_{i_{j+1}}, \forall j \in \{0, \ldots, n-2\}$.

Presupunem din nou că secvenţa $(a_0, \ldots, a_{n-1})$ este memorată în tabloul $(\mathtt{a[i]} \mid 0 \le i \le n-1)$. Algoritmul sortării prin inserare consideră că în pasul $k$, elementele `a[0..k-1]` sunt sortate crescător, iar elementul `a[k]` va fi inserat, astfel încât, după această inserare, primele elemente `a[0..k]` să fie sortate crescător. Inserarea elementului `a[k]` în secvenţa `a[0..k-1]` presupune:

1. memorarea elementului într-o variabilă temporară;
2. deplasarea tuturor elementelor din vectorul `a[0..k-1]` care sunt mai mari decât `a[k]`, cu o poziţie la dreapta (aceasta presupune o parcurgere de la dreapta la stânga);

3. plasarea lui a[k] în locul ultimului element deplasat.

Algoritmul $A_3$ care rezolvă $P_3$ este descris de următorul program:

```
/* algoritmul A3 */
for k ← 1 to n-1 do
   temp ← a[k]
   i  ← k-1
   while (i ≥ 0 and a[i] > temp) do
      a[i+1] ← a[i]
      i  ← i-1
   a[i+1]  ← temp
```

Dimensiunea problemei $P_3$ este dată de numărul $n$ al elementelor din secvenţa ce urmează a fi sortată. Presupunem şi aici că toate operaţiile necesită o unitate de timp. Cazul cel mai favorabil este obţinut când elementele secvenţei sunt sortate crescător. În această situaţie se efectuează $n + 2(n-1)$ comparaţii şi $n + 3(n-1)$ atribuiri. Rezultă $T_{A_1}^{fav}(n) = 3n - 2 + 4n - 3 = 6n - 5$. Cazul cel mai defavorabil este dat de situaţia în care deplasarea (la dreapta cu o poziţie în vederea inserării) se face de la începutul tabloului, adică şirul este ordonat descrescător. Estimarea timpului de execuţie pentru cazul cel mai nefavorabil este următoarea: numărul de comparaţii este $n + 2(2 + 3 + \cdots + n) = n + n(n+1) - 2 = n^2 + 2n - 2$, iar cel al atribuirilor este $n + 3(n-1) + 2(1 + 2 + \cdots + n - 1) = 4n - 3 + (n-1)n = n^2 + 3n - 3$. Adunând, avem $T_{A_3}(n) = (n^2 + 2n - 2) + (n^2 + 3n - 3) = 2n^2 + 5n - 5$. Spaţiul este $n + 5$ (tabloul a, constantele 0 şi 1, variabilele i, temp, n şi z). sfex

**Exemplu.** Căutarea binară. Reconsiderăm problema căutării unui element într-un tablou, dar cu presupunerea suplimentară că tabloul este ordonat.

**Problema** $P_4$

*Intrare:* $n, (a_0, \ldots, a_{n-1}), z$ numere întregi;
secvenţa $(a_0, \ldots, a_{n-1})$ este sortată crescător,
adică $a_i \leq a_{i+1}, i \in \{0, \ldots, n-2\}$.

*Ieşire:* $poz = \begin{cases} k \in \{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

Esenţa căutării binare constă în compararea elementului căutat cu elementul din mijlocul zonei de căutare şi în cazul în care elementul căutat nu este egal cu acesta, se restrânge căutarea la subzona din stânga sau din dreapta, în funcţie de rezultatul comparării. Astfel, dacă elementul căutat este mai mic decât cel din mijlocul zonei de căutare, se alege subzona din stânga, altfel subzona din dreapta. Iniţial, zona de căutare este tabloul a. Convenim să notăm cu $i_{stg}$ indicele elementului din stânga zonei de căutare în tablou, $i_{dr}$ indicele elementului din dreapta zonei de căutare în tablou. Indicele elementului din mijlocul zonei de căutare în tablou va fi notat $i_{med}$.

Algoritmul căutării binare ($A_4$)este descris de subprogramul următor:

```
/* algoritmul A4 */
function cautareBinara(a,n,z)
   istg ← 0
```

```
    i_dr ← n-1
    while (i_stg ≤ i_dr) do
        i_med ← ⌊i_stg+i_dr ⌋/2
        if (a[i_med]=z)
            then return i_med
            else if (a[i_med]>z)
                    then i_dr ← i_med-1 /* se cauta in stanga */
                    else i_stg ← i_med+1 /* se cauta in dreapta */
    return -1
end
```

Dimensiunea problemei $P_4$ este dată de dimensiunea $n$ a secvenţei în care se face căutarea. Şi de această dată presupunem că toate operaţiile necesită o unitate de timp. Calculul timpului de execuţie al algoritmului constă în determinarea numărului de execuţii ale blocului de instrucţiuni asociat cu instrucţiunea `while`. Se observă că, după fiecare iteraţie a buclei `while`, dimensiunea zonei de căutare se înjumătăţeşte. Cazul cel mai favorabil este obţinut când $a\lfloor\frac{n-1}{2}\rfloor = z$ şi se efectuează două comparaţii şi trei atribuiri. Rezultă $T_{A_4}^{fav}(n) = 2 + 3 = 5$. Ca şi la căutarea secvenţială, cazul cel mai nefavorabil este în situaţia în care vectorul $a$ nu conţine valoarea căutată. Pentru simplitate, se consideră $n = 2^k$, unde $k$ este numărul de înjumătăţiri. Rezultă $k = \log_2 n$ şi printr-o majorare, $T_{A_4}(n) \leq c \log_2 n + 1$, unde $c$ este o constantă, $c \geq 1$. Spaţiul necesar execuţiei algoritmului $A_4$ este $n+7$ (tabloul `a`, constantele `0` şi `-1`, variabilele $i_{stg}$, $i_{dr}$, $i_{med}$, `n` şi `z`). sfex

**Observaţie.** Timpii de execuţie pentru cazul cel mai favorabil nu oferă informaţii relevante despre eficienţa algoritmului. Mult mai semnificative sunt informaţiile oferite de timpii de execuţie în cazul cel mai nefavorabil: în toate celelalte cazuri algoritmul va avea performanţe mai bune sau cel puţin la fel de bune.

Pentru evaluarea timpului de execuţie nu este necesar întotdeauna să numărăm toate operaţiile. Pentru exemplul de mai sus, observăm că operaţiile de atribuire (fără cea iniţială) sunt precedate de comparaţii. Astfel, putem număra numai comparaţiile, pentru că numărul acestora determină numărul atribuirilor. Putem să mergem chiar mai departe şi să numărăm numai comparaţiile între $z$ şi componentele tabloului. sfobs

Uneori, numărul instanţelor $p$ cu $g(p) = n$ pentru care $T_A(p) = T_A(n)$ sau $T_A(p)$ are o valoare foarte apropiată de $T_A(n)$ este foarte mic. Pentru aceste cazuri, este preferabil să calculăm comportarea în medie a algoritmului. Pentru a putea calcula comportarea în medie este necesar să privim mărimea $T_A(p)$ ca fiind o variabilă aleatoare (o experienţă = execuţia algoritmului pentru o instanţă $p$, valoarea experienţei = durata execuţiei algoritmului pentru instanţa $p$) şi să precizăm legea de repartiţie a acestei variabile aleatoare. Apoi, *comportarea în medie* se calculează ca fiind media acestei variabile aleatoare (considerăm numai cazul timpului de execuţie):

$$T_A^{med}(n) = M(\{T_A(p) \mid p \in P \wedge g(p) = n\})$$

Dacă mulţimea valorilor variabilei aleatoare $T_A(p) = \{x_1, \ldots\}$ este finită sau numărabilă ($T_A(p) = \{x_1, \ldots, x_i \ldots\}$ şi probabilitatea ca $T_A(p) = x_i$ este $p_i$, atunci

media variabilei aleatoare $T_A$ (timpul mediu de execuție) este:

$$T_A^{med}(n) = \sum_i x_i \cdot p_i$$

**Exemplu.** Considerăm problema $P_1$ definită anterior. Mulțimea valorilor variabilei aleatoare $T_A(p)$ este $\{3i + 2 \mid 1 \leq i \leq n\}$. În continuare trebuie să stabilim legea de repartiție. Facem următoarele presupuneri: probabilitatea ca $z \in \{a_0, \ldots, a_{n-1}\}$ este $q$ și probabilitatea ca $z$ să apară prima dată pe poziția $i-1$ este $\frac{q}{n}$ (indicii $i$ candidează cu aceeași probabilitate pentru prima apariție a lui $z$). Rezultă că probabilitatea ca $z \notin \{a_0, \ldots, a_{n-1}\}$ este $1 - q$. Acum, probabilitatea ca $T_A(p) = 3i + 2$ ($poz = i - 1$) este $\frac{q}{n}$, pentru $1 \leq i < n$, iar probabilitatea ca $T_A(p) = 3n + 2$ este $p_n = \frac{q}{n} + (1 - q)$ (probabilitatea ca $poz = n - 1$ sau ca $z \notin \{a_0, \ldots, a_{n-1}\}$). Timpul mediu de execuție este:

$$\begin{aligned}
T_A^{med}(n) &= \sum_{i=1}^{n} p_i x_i = \sum_{i=1}^{n-1} \frac{q}{n} \cdot (3i + 2) + (\frac{q}{n} + (1 - q)) \cdot (3n + 2) \\
&= \frac{3q}{n} \cdot \sum_{i=1}^{n} i + \frac{q}{n} \sum_{i=1}^{n} 2 + (1 - q) \cdot (3n + 2) \\
&= \frac{3q}{n} \cdot \frac{n(n+1)}{2} + 2q + (1 - q) \cdot (3n + 2) \\
&= \frac{3q \cdot (n+1)}{2} + 2q + (1 - q) \cdot (3n + 2) \\
&= 3n - \frac{3nq}{2} + \frac{3q}{2} + 2
\end{aligned}$$

Pentru $q = 1$ ($z$ apare totdeauna în secvență) avem $T_A^{med}(n) = \frac{3n}{2} + \frac{7}{2}$ și pentru $q = \frac{1}{2}$ avem $T_A^{med}(n) = \frac{9n}{4} + \frac{11}{4}$. sfex

### 1.3.1 Calcul asimptotic

În practică, atât $T_A(n)$, cât și $T_A^{med}(n)$ sunt dificil de evaluat. Din acest motiv se caută, de multe ori, margini superioare și inferioare pentru aceste mărimi. Următoarele clase de funcții sunt utilizate cu succes în stabilirea acestor margini:

$$O(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \leq c \cdot |f(n)|\}$$
$$\Omega(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \geq c \cdot |f(n)|\}$$
$$\Theta(f(n)) = \{g(n) \mid (\exists c_1, c_2 > 0, n_0 \geq 0)(\forall n \geq n_0) c_1 \cdot |f(n)| \leq |g(n)| \leq c_2 \cdot |f(n)|\}$$

Cu notațiile $O$, $\Omega$ și $\Theta$ se pot forma expresii și ecuații. Considerăm numai cazul $O$, celelalte tratându-se similar. Expresiile construite cu $O$ pot fi de forma:

$$O(f_1(n)) \text{ op } O(f_2(n))$$

unde „op" poate fi $+, -, *$ etc. și notează mulțimile:

$$\{g(n) \mid (\exists g_1(n), g_2(n), c > 0, n_0 \geq 0)$$
$$[(\forall n) g(n) = g_1(n) \text{ op } g_2(n)] \wedge [(\forall n \geq n_0) g_1(n) \leq c f_1(n) \wedge g_2(n) \leq c f_2(n)]\}$$

De exemplu:

$$O(n) + O(n^2) = \{g(n) = g_1(n) + g_2(n) \mid (\forall n \geq n_0) g_1(n) \leq cn \wedge g_2(n) \leq cn^2\}$$

Utilizând regulile de asociere şi prioritate, se obţin expresii de orice lungime:

$$O(f_1(n)) \text{ op}_1 \ O(f_2(n)) \text{ op}_2 \ \cdots$$

Orice funcţie $f(n)$ poate fi gândită ca o notaţie pentru mulţimea cu un singur element $f(n)$ şi deci putem alcătui expresii de forma:

$$f_1(n) + O(f_2(n))$$

ca desemnând mulţimea:

$$\{f_1(n) + g(n) \mid g(n) \in O(f_2(n))\} =$$

$$\{f_1(n) + g(n) \mid (\exists c > 0, n_0 > 1)(\forall n \geq n_0) g(n) \leq c \cdot f_2(n)\}$$

Peste expresii considerăm formule de forma:

$$expr1 = expr2$$

cu semnificaţia că mulţimea desemnată de $expr1$ este inclusă în mulţimea desemnată de $expr2$. De exemplu, avem:

$$n \log n + O(n^2) = O(n^2)$$

pentru că $(\exists c_1 > 0, n_1 > 1)(\forall n \geq n_1) n \log n \leq c_1 n^2$, $g_1(n) \in O(n^2)$ implică $(\exists c_2 > 0, n_2 > 1)(\forall n \geq n_2) g_1(n) \leq c_2 n^2$ şi de aici $(\forall n \geq n_0) g(n) = n \log n + g_1(n) \leq n \log n + c_2 n^2 \leq (c_1 + c_2) n^2$, unde $n_0 = \max\{n_1, n_2\}$. De remarcat nesimetria ecuaţiilor: părţile stânga şi cea din dreapta joacă roluri distincte. Ca un caz particular, notaţia $g(n) = O(f(n))$ semnifică, de fapt, $g(n) \in O(f(n))$.

### 1.3.2 Calculul timpului de execuţie pentru cazul cel mai nefavorabil

Un algoritm poate avea o descriere complexă şi deci evaluarea sa poate pune unele probleme. De aceea prezentăm câteva strategii ce sunt aplicate în determinarea timpului de execuţie pentru cazul cel mai nefavorabil. Deoarece orice algoritm este descris de un program, în continuare considerăm $A$ o secvenţă de program. Regulile prin care se calculează timpul de execuţie sunt date în funcţie de structura lui $A$:

1. $A$ este o instrucţiune de atribuire. Timpul de execuţie a lui $A$ este egal cu timpul evaluării expresiei din partea dreaptă.
2. $A$ este forma:

   $A_1$
   $A_2$

   Timpul de execuţie al lui $A$ este egal cu suma timpilor de execuţie ai algoritmilor $A_1$ şi $A_2$.

3. $A$ este de forma `if` $c$ `then` $A_1$ `else` $A_2$. Timpul de execuţie al lui $A$ este egal cu maximul dintre timpii de execuţie ai algoritmilor $A_1$ şi $A_2$ la care se adună timpul necesar evaluării expresiei $c$.
4. $A$ este de forma `while` $c$ `do` $A_1$. Se determină cazul când se execută numărul maxim de iteraţii ale buclei `while` şi se face suma timpilor calculaţi pentru fiecare iteraţie. Dacă nu este posibilă determinarea timpilor pentru fiecare iteraţie, atunci timpul de execuţie al lui $A$ este egal cu produsul dintre timpul maxim de execuţie al algoritmului $A_1$ şi numărul maxim de execuţii ale buclei $A_1$.

Plecând de la aceste reguli de bază, se pot obţine în mod natural reguli de calcul a timpului de execuţie pentru toate instrucţiunile.

**Teorema 1.2.** *Dacă $g$ este o funcţie polinomială de grad $k$, atunci $g = O(n^k)$.*

*Demonstraţie.* Presupunem $g(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots + a_1 \cdot n + a_0$. Efectuând majorări în membrul drept, obţinem:

$$g(n) \leq |a_k| \cdot n^k + |a_{k-1}| \cdot n^{k-1} + \cdots + |a_1| \cdot n + |a_0| < n^k \cdot \underbrace{(|a_k| + |a_{k-1}| + |a_0|)}_{c} < n^k \cdot c$$

pentru $\forall n > 1 \Rightarrow g(n) < c \cdot n^k$, cu $n_0 = 1$. Deci $g = O(n^k)$. sfdem

Notaţiile $O, \Omega$ şi $\Theta$ oferă informaţii cu care putem aproxima comportarea unei funcţii. Pentru ca această aproximare să aibă totuşi un grad de precizie cât mai mare, este necesar ca mulţimea desemnată de partea dreaptă a ecuaţiei să fie cât mai mică. De exemplu, avem atât $3n = O(n)$, cât şi $3n = O(n^2)$. Prin incluziunea $O(n) = O(n^2)$ rezultă că prima ecuaţie oferă o aproximare mai bună. De fiecare dată vom căuta mulţimi care aproximează cel mai bine comportarea unei funcţii.

Următoarele incluziuni sunt valabile în cazul notaţiei $O$:

$$O(1) \subset O(\log n) \subset O(\log^k n) \subset O(n) \subset O(n^2) \subset \cdots \subset O(n^{k+1}) \subset O(2^n)$$

Notaţia $O$ este utilizată şi pentru clasificarea algoritmilor. Cele mai cunoscute clase sunt:

$\{A \mid T_A(n) = O(1)\}$ = clasa algoritmilor constanţi;
$\{A \mid T_A(n) = O(\log n)\}$ = clasa algoritmilor logaritmici;
$\{A \mid T_A(n) = O(log^k n)\}$ = clasa algoritmilor polilogaritmici;
$\{A \mid T_A(n) = O(n)\}$ = clasa algoritmilor liniari;
$\{A \mid T_A(n) = O(n^2)\}$ = clasa algoritmilor pătratici;
$\{A \mid T_A(n) = O(n^k)\}$ = clasa algoritmilor polinomiali;
$\{A \mid T_A(n) = O(2^n)\}$ = clasa algoritmilor exponenţiali.

Cu notaţiile de mai sus, doi algoritmi, care rezolvă aceeaşi problemă, pot fi comparaţi numai dacă au timpii de execuţie în clase de funcţii (corespunzătoare notaţiilor $O, \Omega$ şi $\Theta$) diferite. De exemplu, un algoritm $A$ cu $T_A(n) = O(n)$ este mai eficient decât un algoritm $A'$ cu $T_{A'}(n) = O(n^2)$. Dacă cei doi algoritmi au timpii de execuţie în aceeaşi clasă, atunci compararea lor devine mai dificilă pentru că trebuie determinate şi constantele cu care se înmulţesc reprezentanţii clasei.

Putem acum încadra algoritmii prezentaţi în exemplele anterioare în clasele lor. Astfel, pentru algoritmul ce rezolva problema $P_1$ (căutarea unui element într-o secvenţă de numere), $T_{A_1}(n) = 3n + 2$. Deci, timpul de execuţie al acestui algoritm pentru cazul cel mai nefavorabil este în clasa $O(n)$.

Pentru problema $P_2$ (căutarea într-un şir a elementului de valoare maximă), $T_{A_2}(n) = 1+2(n-1)$. Timpul de execuţie a algoritmului ce rezolvă această problemă pentru cazul cel mai nefavorabil este în clasa $O(n)$.

În cazul problemei $P_3$ (sortarea prin inserare), $T_{A_3}(n) = 3(n-1) + 3n(n-1)/2$. Rezultă că timpul de execuţie al algoritmului respectiv este în clasa $O(n^2)$.

Pentru problema $P_4$ (căutarea binară) nu se poate aplica teorema 1.2. Deoarece $T_{A_4}(n) \leq \log_2 n + 1$, timpul de execuţie al acestui algoritm este în clasa $O(\log_2 n)$. Baza logaritmului se poate ignora deoarece: $\log_a x = \log_a b \log_b x$ şi $\log_a b$ este o constantă, deci rămâne $O(\log n)$, adică o funcţie logaritmică. O demonstraţie riguroasă pentru evaluarea timpului de execuţie al algoritmului căutării binare este dată în capitolul 5, lema 5.1, corolarul 5.1.

### 1.3.3 Exerciţii

**Exerciţiul 1.3.1.** Se consideră programul:

```
x ← 0
y ← 3
while (y > 0) do
    x ← x + 4
    y ← y - 1
```

a) Să se descrie calculul programului.

b) Presupunând că:
   - o comparaţie `x > y`, respectiv o adunare `x + y`, se realizează în timpul $\lfloor \log_2 |x| \rfloor + 1 + \lfloor \log_2 |y| \rfloor + 1$, dacă $x \cdot y \neq 0$, şi timpul 1, dacă $x \cdot y = 0$,
   - o atribuire `x ← a` se realizează în timpul $\lfloor \log_2 |a| \rfloor + 1$, dacă $a \neq 0$, şi timpul 1, dacă $a = 0$,
   - o decrementare `x--` se realizează în timpul $\lfloor \log_2 |x| \rfloor + 1$, dacă $x \neq 0$, şi timpul 1, dacă $x = 0$,

   să se determine timpul necesar executării programului.

c) Ce se poate spune despre timpul de execuţie al algoritmului de înmulţire a două numere prin adunări repetate?

**Exerciţiul 1.3.2.** Se consideră programul:

```
max ← -∞
for i ← 1 to n do
   if a[i] > max
      then max ← a[i]
```

Se presupune ca numerele din $a$ sunt alese arbitrar din $[0, 1]$.

a) Dacă un număr $x$ este ales arbitrar dintr-o mulţime cu $n$ elemente, atunci care este probabilitatea ca $x$ să fie cel mai mare număr din mulţime?

b) Care este probabilitatea ca linia 4 să fie executată o singură dată?

c) Care este probabilitatea ca linia 4 să fie executată de două ori?

**Exerciţiul 1.3.3.** Să se arate că:

1. $n! = O(n^n)$,
2. $n! = \Theta(n^n)$,
3. $5n^2 + n \log n = \Theta(n^2)$,
4. $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$,
5. $\frac{n^k}{\log n} = O(n^k)$ $(k > 0)$,
6. $n^5 + 2^n = O(2^n)$.

**Exerciţiul 1.3.4.** Să se determine timpul de execuţie, pentru cazul cel mai nefavorabil, al algoritmului descris de următorul program:

```
x ← a
y ← b
z ← 1
while (y > 0) do
   if (y impar) then z ← z*x
   y ← y div 2
   x ← x*x
```

Se presupune $a, b \geq 0$. Se vor considera două cazuri:

1. g(p)=b,
2. $g(p) = \lfloor \log n \rfloor$.

*Indicaţie.* Se va ţine cont de faptul că programul calculează $f(a, b) = a^b$ după formula

$$f(u, v) = \begin{cases} 1 & \text{, dacă } v = 0 \\ u^{v-1} * u & \text{, dacă v este impar} \\ (u^{\frac{v}{2}})^2 & \text{, dacă v este par.} \end{cases}$$

**Exerciţiul 1.3.5.** Să se determine timpul de execuţie al algoritmului descris de următorul program:

```
x ← a
y ← b
s ← 0
while x<>0 do
   while not odd(x) do
      y ← 2*y
      x ← x div 2
   s ← s+y
   x ← x-1
```

**Exerciţiul 1.3.6.** Un tablou a[1..$n$] conţine toţi întregii de la 0 la n cu excepţia unuia. Acesta poate fi determinat în timpul $O(n)$ utilizând un tablou suplimentar b[0..$n$] care memorează numerele ce apar în $a$. Presupunem că singura operaţie care permite accesul la elementele tabloului este „extrage bitul $j$ din a[$i$]" care durează o unitate de timp. Să se arate că, utilizând numai această operaţie, este încă posibil să determinăm numărul lipsă în $O(n)$.

**Exerciţiul 1.3.7.** Să se scrie un program care pentru un tablou unidimensional dat, determină dacă toate elementele tabloului sunt egale sau nu. Să se determine timpii de execuţie ai algoritmului descris de program, pentru cazul cel mai nefavorabil şi mediu.

**Exerciţiul 1.3.8.** Se consideră polinomul cu coeficienţi reali $p = a_0 + a_1 x + \cdots + a_n x^n$ şi punctul $x_0$.

a) Să se scrie un program care să calculeze valoarea polinomului $p$ în $x_0$, utilizând formula:

$$p(x_0) = \sum_{i=0}^{n} a_i \cdot x_0^i$$

b) Să se îmbunătăţească programul de mai sus, utilizând relaţia $x_0^{i+1} = x_0^i \cdot x_0$.

c) Să se scrie un program care să calculeze valoarea polinomului $p$ în $x_0$, utilizând formula (schema lui Horner):

$$p(x_0) = a_0 + (\cdots + (a_{n-1} + a_n \cdot x_0) \cdot x_0 \cdots) \cdot x_0$$

Pentru fiecare dintre cele trei cazuri, să se determine numărul de înmulţiri şi de adunări şi să se compare. Care metodă este cea mai eficientă?

**Exerciţiul 1.3.9.** Se consideră programul:

```
k ← rand(n-2) + 1;
for i ← 1 to k do
   a[i] ← rand(n)
max ← -1
while ( i ≤ n and max = -1) do
   a[i] ← rand(n)
   ok ← true
   for j ← 1 to i-1 do
      if a[i] < a[j]
         then ok ← false
   if (ok) then max ← a[i]
if (max = -1) then max ← a[n]
```

a) Când sunt şansele maxime ca `max` să memoreze după execuţia programului cea mai mare valoare din tabloul `a`?

b) Care este timpul mediu de execuţie dacă se numără numai atribuirile de pe penultima linie?

## 1.4 Referinţe bibliografice

Temele abordate în acest capitol se regăsesc în toate cărţile dedicate studierii algoritmilor [Sed88, CLR93, CLR00, Baa78, BG00, Mel84a, AHU74, HS84, MB00, MS91, LG86].

# Capitolul 2

# Tipuri de date de nivel înalt

După cum am precizat în capitolul 1, tipurile de date de nivel înalt pot fi descrise într-o manieră independentă de limbaj. În aceasta categorie am inclus: listele liniare, stivele, cozile, listele generalizate, arborii binari, cozile cu priorităţi, grafurile, *union-find*.

## 2.1 Liste liniare

### 2.1.1 Tipul de date abstract LLIN

#### 2.1.1.1 Descrierea obiectelor

O *listă liniară* este o secvenţă finită $(e_0, \ldots, e_{n-1})$ în care elementele $e_i$ aparţin unui tip dat Elt. Componentele unei liste nu trebuie să fie neapărat distincte; adică, putem avea $i \neq j$ şi $e_i = e_j$. *Lungimea* unei liste $L = (e_0, \ldots, e_{n-1})$, notată cu Lung$(L)$, este dată de numărul de componente din listă: Lung$(L) = n$. *Lista vidă* ( ) este lista fără nici o componentă (de lungime 0).

#### 2.1.1.2 Operaţii

ListaVidă. Întoarce lista vidă.

*Intrare:* – nimic;
*Ieşire:* – lista vidă ( ).

EsteVidă. Testează dacă o listă dată este vidă.

*Intrare:* – o listă liniară $L$;
*Ieşire:* – *true* dacă $L = ( )$,
– *false* dacă $L \neq ( )$.

Lung. Întoarce lungimea unei listei date.

*Intrare:* – o listă liniară $L = (e_0, \ldots, e_{n-1})$ cu $n \geq 0$;
*Ieşire:* – $n$.

Copie. Întoarce o copie distinctă a unei liste date.

*Intrare:* – o listă liniară $L$;
*Ieşire:* – o copie $L'$ distinctă a lui $L$.

Egal. Testează dacă două liste liniare sunt egale.

*Intrare:* – două liste $L = (e_0, \ldots, e_{n-1})$ şi $L' = (e'_0, \ldots, e'_{n'-1})$;
*Ieşire:* – *true* dacă $n = n'$ şi $e_0 = e'_0, \ldots, e_{n-1} = e'_{n-1}$,
– *false*, în celelalte cazuri.

Poz. Caută dacă un element dat $x$ apare într-o listă liniară dată $L$.

*Intrare:* – o listă liniară $L = (e_0, \ldots, e_{n-1})$ şi un element $x$ din Elt;
*Ieşire:* – adresă invalidă ($-1$ sau NULL) dacă $x$ nu apare în $L$,
– adresa elementului $e_i$ dacă $e_i = x$ şi $(\forall j) 0 \leq j < i \Rightarrow e_j \neq x$.

Parcurge. Parcurge o listă liniară dată efectuând o prelucrare uniformă asupra componentelor acesteia.

*Intrare:* – o listă liniară $L$ şi o procedură Vizitează(e);
*Ieşire:* – lista $L$, dar ale cărei componente au fost prelucrate de procedura Vizitează.

Citeşte. Întoarce elementul de la o poziţie dată dintr-o listă dată.

*Intrare:* – o listă liniară $L = (e_0, \ldots, e_{n-1})$ şi o poziţie $k \geq 0$;
*Ieşire:* – $e_k$ dacă $0 \leq k < n$,
– *eroare*, în celelalte cazuri.

Inserează. Adaugă un element dat într-o listă liniară dată la o poziţie dată.

*Intrare:* – o listă liniară $L = (e_0, \ldots, e_{n-1})$, un element $e$ din Elt şi o poziţie $k \geq 0$;
*Ieşire:* – $L = (e_0, \ldots, e_{k-1}, e, e_k, \ldots, e_{n-1})$ dacă $0 \leq k < n$,
– $L = (e_0, \ldots, e_{n-1}, e)$ dacă $k \geq n$.

EliminăDeLaK. Elimină elementul de la o poziţie dată dintr-o listă liniară dată.

*Intrare:* – o listă liniară $L = (e_0, \ldots, e_{n-1})$ şi o poziţie $k \geq 0$;
*Ieşire:* – $L = (e_0, \ldots, e_{k-1}, e_{k+1}, \ldots, e_{n-1})$ dacă $0 \leq k < n$,
– $L = (e_0, \ldots, e_{n-1})$ dacă $k \geq n$.

Elimină. Elimină un element dat dintr-o listă liniară dată.

*Intrare:* – o listă liniară $L = (e_0, \ldots, e_{n-1})$ şi un element $e$ din Elt;
*Ieşire:* – lista $L$ din care s-au eliminat toate elementele $e_i$ egale cu $e$.

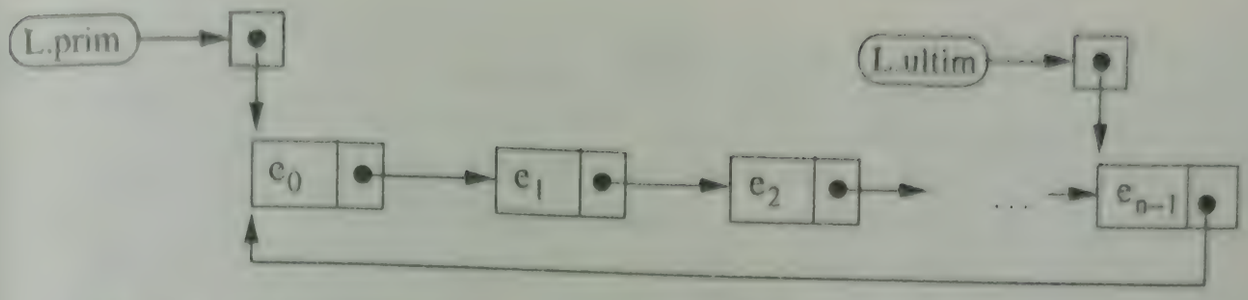



Figura 2.1: Reprezentarea listei liniare cu structură simplu înlănţuită

## 2.1.2 Implementarea cu structuri dinamice simplu înlănţuite

### 2.1.2.1 Reprezentarea obiectelor

O listă liniară $L = (e_0, \ldots, e_{n-1})$ poate fi reprezentată printr-o structură dinamică simplu înlănţuită (figura 2.1). Fiecare componentă a listei liniare este memorată într-un nod al structurii dinamice simplu înlănţuite. Există doi pointeri `L.prim` şi `L.ultim` care fac referire la primul şi respectiv ultimul nod.

### 2.1.2.2 Implementarea operaţiilor

ListaVidă. Lista este reprezentată de structura simplu înlănţuită fără nici un nod. Funcţia ListaVidă întoarce `L.prim` = `L.ultim` = `NULL`

EsteVidă. Se testează dacă pointerul `L.prim` este egal cu `NULL`.

Lung. Parcurge structura simplu înlănţuită şi contorizează fiecare nod parcurs. Timpul necesar determinării lungimii este $O(n)$. Acesta poate fi redus la $O(1)$ dacă lungimea ar fi inclusă în reprezentarea listei. Aceasta ar presupune ca toate operaţiile care modifică lungimea listei să actualizeze corespunzător variabila responsabilă cu memorarea lungimii.

```
function lung(L)
   lg ← 0
   p ← L.prim
   while (p ≠ NULL) do
      lg ← lg+1
      p ← p->succ
   return lg
end
```

Copie. Parcurge structura simplu înlănţuită sursă şi adaugă la structura simplu înlănţuită destinaţie o copie a fiecărui nod parcurs. Iniţial lista destinaţie este vidă. Dacă presupunem că operaţia de copiere a unui nod se face în timpul $O(t)$, atunci copierea întregii liste se realizează în timpul $O(n \cdot t)$.

```
procedure copie (L, Lcopie)
   if (L.prim = NULL)
```

```
        then Lcopie.prim ← NULL
             Lcopie.ultim ← NULL
        else new(Lcopie.prim)
             Lcopie.prim->elt ← L.prim->elt
             Lcopie.ultim ← Lcopie.prim
             p ← L.prim->succ
             while (p ≠ NULL) do
                new(Lcopie.ultim->succ)
                Lcopie.ultim ← Lcopie.ultim->succ
                Lcopie.ultim->elt ← p->elt
                p ← p->succ
             Lcopie.ultim->succ ← NULL
end
```

Egal. Parcurge simultan cele două structuri simplu înlănţuite şi compară perechile de noduri corespunzătoare. Dacă presupunem că operaţia de comparare a unei perechi de noduri se face în timpul $O(1)$, atunci compararea listelor se realizează în timpul $O(n)$ în cazul cel mai nefavorabil, unde $n$ este valoarea minimă dintre lungimile celor două liste.

```
function egal(L1, L2)
   p1 ← L1.prim
   p2 ← L2.prim
   while ((p1 ≠ NULL) and (p2 ≠ NULL)) do
      if (p1->elt ≠ p2->elt) then return false
      p1 ← p1->succ
      p2 ← p2->succ
   if ((p1 = NULL) and (p2 = NULL))
      then return true
      else return false
end
```

Poz. Se aplică tehnica căutării secvenţiale: se pleacă din primul nod şi se interoghează nod cu nod până când este găsit primul care îl memorează pe $x$ sau au fost epuizate toate nodurile. Plecând de la ipoteza că interogarea unui nod se face în timpul $O(1)$, rezultă că operaţia de căutare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
function poz(L, x)
   p ← L.prim
   while (p ≠ NULL) do
      if (p->elt = x) then return p
      p ← p->succ
   return NULL
end
```

Parcurge. Se parcurge structura simplu înlănțuită nod cu nod și se aplică procedura Vizitează. Se obține un grad mai mare de aplicabilitate a operației dacă procedurile de tip Vizitează au ca parametru adresa unui nod și nu informația memorată în nod. Dacă $t$ este timpul necesar procedurii Vizitează pentru a prelucra un nod, atunci timpul de execuție al operației de parcurgere este în clasa $O(n \cdot t)$.

```
procedure parcurge(L, viziteaza())
   p ← L.prim
   while (p ≠ NULL) do
      viziteaza(p)
      p ← p->succ
end
```

Citește. Pentru a putea citi informația din cel de-al $k$-lea nod, este necesară parcurgerea secvențială a primelor $k$-noduri. Deoarece $k < n$, rezultă că timpul de execuție în cazul cel mai nefavorabil este în clasa $O(n)$.

```
function citeste(L, k)
   if (k<0) then throw 'eroare'
   p ← L.prim
   i ← 0
   while ((p ≠ NULL) and (i < k)) do
      i ← i + 1
      p ← p->succ
   if (p = NULL) then throw 'eroare'
   return p->elt
end
```

Inserează. Ca și în cazul operației de citire, este necesară parcurgerea secvențială a primelor $k-1$ noduri (noul nod va fi al $k$-lea). Dacă avem $k = 0$ sau $k \geq \text{Lung}(L)$, atunci pointerii `prim` și respectiv `ultim` se actualizează corespunzător. Deoarece operația de adăugare a unui nod după o adresă dată se realizează în timpul $O(1)$, rezultă că operația de inserare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
procedure insereaza(L, k, e)
   if (k < 0) then throw 'eroare'
   new(q)
   q->elt ← e
   if ((k = 0) or (L.prim = NULL))
      then q->succ ← L.prim
           L.prim ← q
           if (L.ultim = NULL) then L.ultim ← q
      else p ← L.prim
           i ← 0
           while ((p ≠ L.ultim) and (i < k-1)) do
              i ← i + 1
              p ← p->succ
           q->succ ← p->succ
```

```
            p->succ ← q
            if (p = L.ultim) then L.ultim ← q
end
```

ElimināDeLaK. Se realizează într-o manieră asemănătoare cu cea a operaţiei de inserare.

```
procedure eliminaDeLaK(L, k)
   if ((k < 0) or (L.prim = NULL)) then throw 'eroare'
   if (k = 0)
      then p ← L.prim
           L.prim ← L.prim->succ
           if (L.ultim = p) then L.ultim ← NULL
           delete(p)
      else p ← L.prim
           i ← 0
           while ((p->succ ≠ NULL) and (i < k-1)) do
              i ← i + 1
              p ← p->succ
           if (p->succ ≠ NULL)
              then q ← p->succ
                   p->succ ← q->succ
                   if (L.ultim = q) then L.ultim ← p
                   delete(q)
end
```

Elimină. Se parcurge structura simplu înlănţuită secvenţial şi fiecare nod care memorează un element egal cu $e$ este eliminat. Eliminarea unui nod presupune cunoaşterea adresei nodului anterior; aceasta este determinată în timpul parcurgerii secvenţiale. Dacă se elimină primul sau ultimul nod atunci pointerii `L.prim` şi respectiv `L.ultim` se actualizează corespunzător. Operaţia de eliminare a unui nod se realizează în timpul $O(1)$. Dacă operaţia de comparare se realizează în timpul $O(1)$ atunci operaţia de eliminare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
procedure elimina(L, e)
   while ((L.prim ≠ NULL) and (L.prim->elt = e)) do
      q ← L.prim
      L.prim ← q->succ
      delete(q)
   if (L.prim = NULL)
      then L.ultim ← NULL
      else p ← L.prim
           while (p ≠ NULL) do
              q ← p->succ
              if ((q ≠ NULL) and (q->elt = e))
                 then p->succ ← q->succ
                      if (L.ultim = q) then L.ultim ← p
```

```
                delete(q)
            p ← p->succ
end
```

## 2.1.3 Implementarea cu tablouri

### 2.1.3.1 Reprezentarea obiectelor

În afara reprezentării prin structuri simplu înlănţuite, o listă liniară $L = (e_0, \ldots, e_{n-1})$ mai poate fi reprezentată şi printr-un tablou `L.tab` şi o variabilă de tip indice `L.ultim` ca în figura 2.2. Cele $n$ componente ale listei liniare sunt memorate în primele $n$ componente ale tabloului. Dimensiunea maximă a tabloului este $MAX$ astfel că vor putea fi reprezentate numai liste de lungime $\leq MAX$. Variabila `L.ultim` memorează indicele (adresa) în tablou a ultimului element din listă.

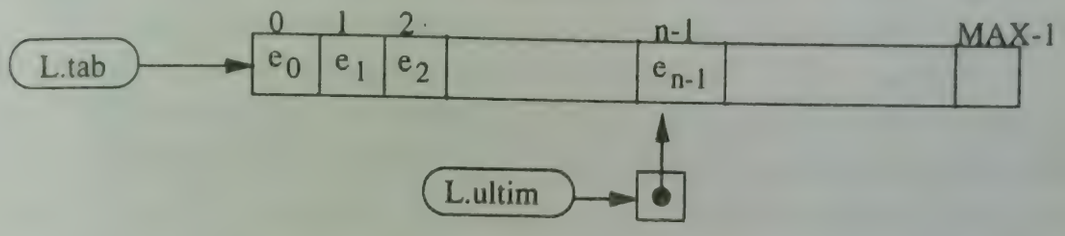



Figura 2.2: Reprezentarea listei liniare cu tablouri

### 2.1.3.2 Implementarea operaţiilor

ListaVidă. Dacă valoarea variabilei `L.ultim` este $-1$, atunci lista este considerată ca fiind vidă.

EsteVidă. Se testează dacă valoarea variabilei `L.ultim` este egală cu $-1$.

Lung. Întoarce valoarea variabilei `L.ultim` adunată cu 1. Deci operaţia este realizată în timpul $O(1)$.

```
function lung(L)
   return L.ultim+1
end
```

Copie. Parcurge tabloul sursă şi copie fiecare componentă parcursă. Dacă presupunem că operaţia de copiere a unui nod se face în timpul $O(t)$, atunci copierea întregii liste se realizează în timpul $O(n \cdot t)$.

```
procedure copie(L, Lcopie)
   Lcopie.ultim ← L.ultim
   for i ← 0 to L.ultim do
       Lcopie.tab[i] ← L.tab[i]
end
```

Egal. Parcurge simultan cele două tablouri şi compară perechile de componente corespunzătoare. Dacă presupunem că operaţia de comparare a unei perechi de elemente se face în timpul $O(1)$ atunci compararea listelor se realizează în timpul $O(n)$ în cazul cel mai nefavorabil, unde $n$ este valoarea minimă dintre lungimile celor două liste.

```
function egal(L1, L2)
   if (L1.ultim ≠ L2.ultim)
      then return false
      else for i ← 0 to L1.ultim do
                if (L1.tab[i] ≠ L2.tab[i]) then return false
           return true
end
```

Poz. Se aplică tehnica căutării secvenţiale: se pleacă din prima componentă a tabloului şi se interoghează componentă cu componentă până când este găsită prima care îl memorează pe $x$ sau au fost epuizate toate componentele de indice $\leq$ L.ultim. Plecând de la ipoteza că interogarea unei componente se face în timpul $O(1)$ rezultă că operaţia de căutare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
function poz(L, x)
   i ← 0
   while ((i < L.ultim) and (L.tab[i] ≠ x)) do
      i ← i+1
   if (L.tab[i] = x)
      then return i
      else return -1
end
```

Parcurge. Se parcurge tabloul componentă cu componentă şi se aplică procedura Vizitează. Ca şi în cazul implementării cu structuri dinamice simplu înlănţuite, se obţine un grad mai mare de aplicabilitate a operaţiei dacă procedurile de tip Vizitează au ca parametru adresa în tablou. Dacă $t$ este timpul necesar procedurii Vizitează pentru a prelucra un nod, atunci timpul de execuţie al operaţiei de parcurgere este $O(n \cdot t)$.

```
procedure parcurge(L, viziteaza())
   for i ← 0 to L.ultim do
       viziteaza(L.tab, i)
end
```

Citeşte. Întoarce valoarea din componenta $k$ a tabloului. Este realizată în timpul $O(1)$.

```
function citeste(L, k)
   if ((k >L.ultim) or (k < 0)) then throw 'eroare'
   return L.tab[k]
end
```

**Inserează.** Elementele memorate in componentele $k, k+1, \ldots, ultim$ trebuie deplasate la dreapta cu o poziţie pentru a elibera componenta $k$. Valoarea variabilei `L.ultim` este incrementată cu o unitate. Cum $k$ poate fi şi 0, rezultă că operaţia de inserare necesită timpul $O(n)$ în cazul cel mai nefavorabil.

```
procedure insereaza(L, k, e)
   if (k < 0) then throw 'eroare'
   if (L.ultim = MAX-1) then throw 'memorie insuficienta'
   L.ultim ← L.ultim+1
   if (k ≥ L.ultim)
      then L.tab[L.ultim] ← e
      else for i ← L.ultim downto k+1 do
               L.tab[i] ← L.tab[i-1]
           L.tab[k] ← e
end
```

**EliminăDeLaK.** Elementele memorate în componentele $k, k+1, \ldots, ultim$ trebuie deplasate la stânga cu o poziţie. Valoarea variabilei `L.ultim` este decrementată cu o unitate. Timpul de execuţie în cazul cel mai nefavorabil este $O(n)$.

```
procedure eliminaDeLaK(L, k)
   if (k < 0) then throw 'eroare'
   if (L.ultim = -1) then throw 'lista vida'
   L.ultim ← L.ultim-1
   for i ← k to L.ultim do
       L.tab[i] ← L.tab[i+1]
end
```

**Elimină.** Se parcurge tabloul secvenţial şi fiecare componentă care memorează un element egal cu $e$ este eliminată prin deplasarea la stânga a elementelor memorate după ea. La fiecare eliminare, variabila `L.ultim` se actualizează corespunzător. Operaţia de eliminare a unui nod se realizează în timpul $O(n-k)$, unde $k$ este indicele în tablou al componentei eliminate. Dacă operaţia de comparare se realizează în timpul $O(1)$, atunci operaţia de eliminare necesită timpul $O(n^2)$ în cazul cel mai nefavorabil (când toate elementele sunt egale cu $e$).

```
procedure elimina(L, e)
   k ← 0
   while (k ≤ L.ultim) do
      if (L.tab[k] = e)
         then L.ultim ← L.ultim-1
              for i ← k to L.ultim do
                  L.tab[i] ← L.tab[i+1]
end
```

### 2.1.4 Exerciţii

**Exerciţiul 2.1.1.** Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea polinoamelor rare cu o singură variabilă şi cu coeficienţi întregi (polinoame de grade mari cu mulţi coeficienţi egali cu zero). Lista va memora numai coeficienţii $\neq 0$. Să se scrie proceduri care, utilizând această structură, realizează operaţiile algebrice cu polinoame şi operaţiile de citire şi scriere de polinoame.

**Exerciţiul 2.1.2.** Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea întregilor mari într-o bază $b$. Să se scrie proceduri care realizează operaţii aritmetice cu întregi reprezentaţi astfel.

**Exerciţiul 2.1.3.** Fie $A$ un alfabet. Dacă $w = a_1 \dots a_n$ este un şir din $A^*$, atunci prin $\overline{w}$ notăm oglinditul lui $w$, $\overline{w} = a_n \dots a_1$. Presupunem că şirurile din $A^*$ sunt reprezentate prin liste liniare simplu înlănţuite. Să se scrie subprograme care să realizeze:

- fiin dat un şir $w$, determină oglinditul său $\overline{w}$;
- fiind dat un şir $w$, decide dacă $w$ este de forma $u\overline{u}$;
- fiind date două şiruri $w$ şi $w'$, decide dacă $w \leq w'$ (se consideră ordinea lexicografică).

**Exerciţiul 2.1.4.** Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea mulţimilor finite de numere complexe. Să se scrie proceduri care să realizeze operaţii cu mulţimi de numere complexe reprezentate astfel.

**Exerciţiul 2.1.5.** Să se proiecteze o structură de date de tip listă liniară pentru reprezentarea numerelor raţionale mari şi cu multe zecimale. Să se scrie proceduri care să realizeze operaţii aritmetice cu numere raţionale reprezentate astfel.

**Exerciţiul 2.1.6.** Se consideră un şir de numere întregi memorat într-un tablou unidimensional a:

- Să se plaseze numerele din tabloul unidimesional a într-o listă liniară liniară implementată cu structuri dinamice simplu înlanţuite, prin inserări repetate în faţa listei;
- Se dă o valoare întreagă $x$. Să se determine dacă $x$ se află printre elementele listei create;
- Să se insereze în listă pe o poziţie dată, un element care să conţină valoarea $x$;
- Să se determine elementul situat pe poziţia k, numărată de la sfârşitul la începutul listei, fără a parcurge lista mai mult de o dată.

**Exerciţiul 2.1.7.** Fie $X = (x_0, x_1, ..., x_{n-1})$ şi $Y = (y_0, y_1, ..., y_{m-1})$ două liste liniare implementate cu structuri dinamice simplu înlanţuite. Să se scrie un program care să realizeze:

- concatenarea celor două liste: $Z = (x_0, x_1, .., x_{n-1}, y_0, y_1, ..., y_{m-1})$;
- interclasarea celor două liste astfel:
  - $Z = (x_0, y_0, x_1, y_1, ..., x_{m-1}, y_{m-1}, x_m, ..., x_{n-1})$ dacă $m < n$,
  - $Z = (x_0, y_0, x_1, y_1, ..., x_{n-1}, y_{n-1}, y_n, ..., y_{m-1})$ dacă $n <= m$.

**Exerciţiul 2.1.8.** Se consideră o listă liniară implementată cu structură dinamică simplu înlanţuită. Să se scrie subprograme care implementează următoarele operaţii

- parcurgerea listei în ambele sensuri (dus-întors) utilizând $O(1)$ memorie suplimentară;
- testarea situaţiei în care o listă are bucle;
- determinarea mijlocului listei (poziţia elementului situat în mijlocul listei).

## 2.2 Liste liniare ordonate

### 2.2.1 Tipul de date abstract LLINORD

#### 2.2.1.1 Descrierea obiectelor

O *listă liniară ordonată* este o listă liniară $(e_0, \ldots, e_{n-1})$, în care elementele $e_i$ aparţin unui tip de date total ordonat Elt şi succesiunea elementelor în secvenţă este în ordine crescătoare: $e_0 < \cdots < e_{n-1}$.

#### 2.2.1.2 Operaţii

Operaţiile ListaVidă, EsteVidă, Lung, Egal, Copie, Parcurge şi Poz sunt aceleaşi ca la lista liniară.

Inserează. Adaugă un element dat într-o listă liniară ordonată astfel încât. după inserare, lista rămâne ordonată crescător.

*Intrare:* – o listă liniară ordonată $L = (e_0 < \cdots < e_{n-1})$ şi un element $e$ din Elt;
*Ieşire:* – $L = (e_0 < \cdots < e_{k-1} < e < e_k < \cdots < e_{n-1})$.

Elimină. Elimină un element dat dintr-o listă liniară ordonată dată.

*Intrare:* – o listă liniară ordonată $L = (e_0 < \cdots < e_{n-1})$ şi un element $e$ din Elt;
*Ieşire:* – $L = (e_0 < \cdots < e_{k-1} < e_{k+1} < \cdots < e_{n-1})$ dacă $e$ apare în $L$ pe locul $k$ $(e = e_k)$,
– lista $L$ neschimbată dacă $e$ nu apare în $L$.

### 2.2.2 Implementarea cu structuri dinamice simplu înlănţuite

#### 2.2.2.1 Reprezentarea obiectelor

Similară cu cea a listelor liniare.

#### 2.2.2.2 Implementarea operaţiilor

Operaţiile ListaVidă, EsteVidă, Lung, Egal, Copie şi Parcurge au aceleaşi implementări precum cele de la lista liniară.

Poz. Algoritmul de căutare secvenţială poate fi îmbunătăţit în sensul următor: dacă s-a întâlnit un element $e_k > x$ atunci toate elementele care urmează sunt mai mari decât $x$ şi procesul de căutare se termină fără succes (întoarce valoarea $-1$). Timpul de execuţie în cazul cel mai nefavorabil rămâne $O(n)$.

```
function poz(L, x)
begin
   p ← L.prim
   while (p ≠ NULL) do
      if (p->elt = x) then return p
      if (p->elt > x) then return NULL
      p ← p->succ
   return NULL
end
```

Inserează. Se disting următoarele cazuri:

1. Lista $L$ este vidă. Se crecază un nod în care va fi memorat $e$. Pointerii `L.prim` şi `L.ultim` vor referi în continuare acest nod.
2. $e <$ *L.prim->elt*. Se adaugă un nod la începutul listei şi se memorează $e$ în acest nod. Pointerul `L.prim` va referi în continuare acest nod.
3. $e >$ *L.ultim->elt*. Se adaugă un nod la sfârşitul listei şi se memorează $e$ în acest nod. Pointerul `L.ultim` va referi în continuare acest nod.
4. *L.prim->elt* $\leq e \leq$ *L.ultim->elt*. Algoritmul de inserare are două subetape:
   4.1. Caută poziţia pe care trebuie memorat $e$. Dacă în timpul căutării este întâlnit un $e_k = e$, atunci procesul de inserare se termină fără a modifica lista.
   4.2. Dacă procesul de căutare s-a terminat pe prima poziţie $k$ cu proprietatea $e > e_k$, atunci inserează un nod în faţa celui pe care s-a terminat procesul de căutare (acesta memorează cel mai mic element mai mare decât $e$) şi memorează $e$ în acest nou nod.

Timpul de execuţie în cazul cel mai nefavorabil al algoritmului este $O(n)$.

```
procedure insereaza(L, e)
   new(q)
   q->elt ← e
   switch
      case (L.prim = NULL)
         q->succ ← NULL /* cazul 1 */
         L.prim ← q
         L.ultim ← q
      case (e < L.prim->elt)
         q->succ ← L.prim /* cazul 2 */
         L.prim ← q
      case (e > L.ultim->elt)
         q->succ ← NULL /* cazul 3 */
         L.ultim->succ ← q
         L.ultim ← q
```

```
        otherwise /* (L.prim->elt <= e <= Lultim->elt)* /
           p ← L.prim /* cazul 4 */
           while (p->elt < e) do
              p ← p->succ
           if (p->elt ≠ e)
              then q->succ ← p->succ /* subcazul 4.2 */
                   p->succ ← q
                   q->elt ← p->elt
                   p->elt ← e
              else delete(q)
end
```

Elimină. Caută nodul care memorează $e$ ca la implementarea operaţiei Poz. În timpul procesului de parcurgere şi căutare se memorează într-o variabilă pointer adresa nodului ce precedă nodul curent (predecesorul). Dacă a fost găsit un astfel de nod ($e$ apare în listă), atunci îl elimină. Dacă $e$ apare pe prima sau ultima poziţie, atunci trebui actualizaţi pointerii `L.prim` şi respectiv `L.ultim`. Timpul de execuţie în cazul cel mai nefavorabil al algoritmului este $O(n)$.

```
procedure elimina(L, e)
begin
   if (L.prim = NULL) then throw 'eroare'
   p ← L.prim
   if (p->elt = e)
      then L.prim ← p->succ
           delete(p)
      else while ((p ≠ NULL) and (p->elt ≠ e)) do
              predp ← p
              p ← p->succ
           if (p ≠ NULL)
           then predp->succ ← p->succ
                if (p = L.ultim) then L.ultim ← predp
                delete(p)
end
```

## 2.2.3 Implementarea cu tablouri

### 2.2.3.1 Reprezentarea obiectelor

Similară cu cea a listelor liniare.

### 2.2.3.2 Implementarea operaţiilor

Operaţiile ListaVidă, EsteVidă, Lung, Egal, Copie şi Parcurge au aceleaşi implementări ca de la lista liniară.

Poz. **(Căutarea binară)**. Algoritmul este prezentat la pagina 33. Cititorul este invitat să rescrie acel algoritm în termenii listei liniare ordonate. Timpul de execuţie în cazul cel mai nefavorabil al algoritmului este $O(\log_2 n)$. Demonstraţia acestui rezultat este prezentată şi în secţiunea 5.2.

Inserează. Se realizează în doi paşi:

1. Se caută binar poziţia $k$ unde urmează să fie memorat $e$.
2. Dacă în timpul căutării este întâlnită o componentă ce conţine $e$ ($e$ este deja în listă), atunci operaţia se termină fără a modifica lista. Altfel, se translatează la dreapta toate elementele de la poziţia $k$ la poziţia $L.ultim$ şi se memorează $e$ pe poziţia $k$.

Etapa de căutare necesită $O(\log_2 n)$, timp în cazul cel mai nefavorabil, dar operaţia de translatare necesită $O(n)$. Astfel, algoritmul care realizează operaţia de inserare are timpul de execuţie în cazul cel mai nefavorabil $O(n)$.

```
procedure insereaza(L, e)
   p ← 0
   q ← L.ultim
   k ← ⌊(p + q)/2⌋
   while ((e ≠ L.tab[k]) and (p < q)) do
      if (e < L.tab[k])
         then q ← k-1
         else p ← k+1
      k ← ⌊(p + q)/2⌋
   if (e ≠ L.tab[k])
      then if (e > L.tab[k]) then k ← k+1
           if (L.ultim = MAX) then throw 'memorie insuficienta'
           L.ultim ← L.ultim+1
           for i ← L.ultim downto k
               L.tab[i] ← L.tab[i-1]
           L.tab[k] ← e
end
```

Elimină. Se realizează în doi paşi:

1. Se caută binar poziţia $k$ unde este memorat $e$. Dacă Poz $e$ nu apare în listă şi operaţia se termină fără a modifica lista.
2. Se translatează la stânga toate elementele de la poziţia $k+1$ la poziţia $L.ultim$. Ca şi în cazul inserării, algoritmul care realizează operaţia de eliminare are timpul de execuţie în cazul cel mai nefavorabil $O(n)$.

```
procedure elimina(L, e)
   p ← 0
   q ← L.ultim
   k ← ⌊(p + q)/2⌋
   while ((e ≠ L.tab[k]) and (p < q)) do
      if (e < L.tab[k])
```

```
            then q ← k-1
            else p ← k+1
        k ← ⌊(p+q)/2⌋
    if (e = L.tab[k])
        then L.ultim ← L.ultim-1
             for i ← k to L.ultim
                 L.tab[i] ← L.tab[i+1]
end
```

### 2.2.4 Exerciţii

**Exerciţiul 2.2.1.** Se consideră un şir de numere întregi memorat într-un tablou unidimensional a. Să se plaseze valorile din a într-o listă liniară ordonată.

**Exerciţiul 2.2.2.** Să se scrie un program care să creeze o listă liniară ordonată crescător cu toate numerele de forma $2^i \cdot 3^j \cdot 5^k$ ($i, j, k$ întregi pozitivi) mai mici decât un $n$ dat. (În legătură cu problema lui Hamming [Luc93].)

**Exerciţiul 2.2.3.** Fie $X = (x_0, x_1, ..., x_{n-1})$ şi $Y = (y_0, y_1, ..., y_{m-1})$ două liste liniare ordonate crescător. Să se scrie un subprogram care realizează interclasarea celor două liste astfel încât lista obţinută să fie ordonată crescător.

## 2.3 Liste circulare

### 2.3.1 Tipul de date abstract LCIRC

#### 2.3.1.1 Descrierea obiectelor

O *listă circulară* este o secvenţă finită $(e_0, \ldots, e_{n-1})$ în care elementele $e_i$ aparţin unui tip dat Elt iar ordinea indicilor $0, \ldots, n-1$ este dată de regula $i < i+1 \pmod n$, $i = 0, \ldots, n-1$. Altfel spus, o listă circulară este o listă liniară modificată astfel încât dupa $e_{n-1}$ urmează $e_0$.

#### 2.3.1.2 Operaţii

Operaţiile sunt aceleaşi ca la lista liniară.

### 2.3.2 Implementarea cu structuri dinamice simplu înlănţuite

#### 2.3.2.1 Reprezentarea obiectelor

O listă circulară $C = (e_0, \ldots, e_{n-1})$ poate fi reprezentată printr-o structură dinamică simplu înlănţuită în care succesorul ultimului element este primul element. (figura 2.3). Fiecare componentă a listei circulare este memorată într-un nod al structurii dinamice simplu înlănţuite. Ultimul nod al structurii „cunoaşte" adresa primului nod. Există doi pointeri `L.prim` şi `L.ultim` care fac referire la primul şi respectiv ultimul nod. Se poate renunţa la pointerul `L.prim`, deoarece `L.prim` =

`L.ultim->succ`. Aceasta este de fapt unul dintre motivele introducerii structurii de listă circulară.

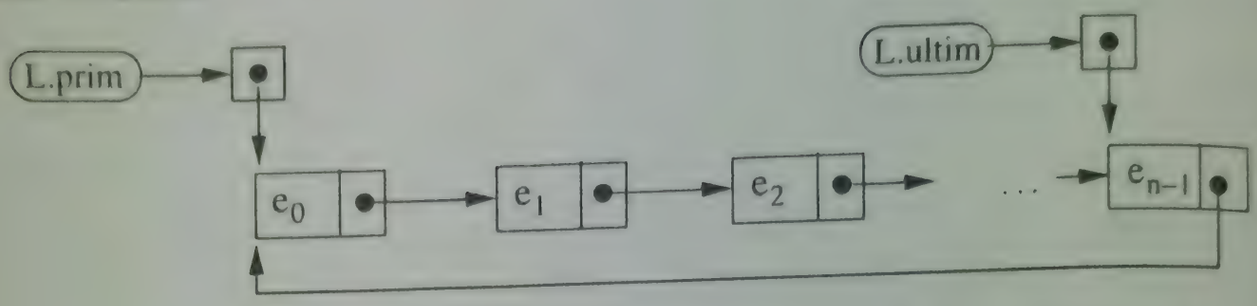



Figura 2.3: Reprezentarea listei circulare cu structură simplu înlănţuită

#### 2.3.2.2 Implementarea operaţiilor

Invităm cititorul să modifice implementările de la liste liniare pentru a obţine algoritmii corespunzători operaţiilor cu liste circulare implementate cu structuri dinamice simplu înlănţuite.

### 2.3.3 Exerciţii

**Exerciţiul 2.3.1.** Se consideră un şir de numere întregi memorat într-un tablou unidimensional `a`.

- Să se scrie un subprogram care să plaseze valorile din `a` într-o listă circulară, prin inserări repetate în faţa listei.
- Se dă o valoare $x$. Să se scrie un subprogram care să determine dacă aceasta se află printre elementele listei create.
- Să se scrie un subprogram care să insereze un element ce conţine valoarea $x$, într-o poziţie dată în listă.
- Să se scrie un subprogram care să şteargă un element din listă, poziţia de ştergere fiind dată.

**Exerciţiul 2.3.2.** Să se scrie un subprogram care inversează legăturile într-o listă circulară implementată cu structură dinamică simplu înlănţuită.

**Exerciţiul 2.3.3.** Fie $X = (x_0, x_1, ..., x_{n-1})$ şi $Y = (y_0, y_1, ..., y_{m-1})$ două liste circulare. Să se scrie un program care realizează:

- concatenarea celor două liste: $Z = (x_0, x_1, ..., x_{n-1}, y_0, y_1, ..., y_{m-1})$;
- interclasarea celor două liste astfel:
  - $Z = (x_0, y_0, x_1, y_1, ..., x_{m-1}, y_{m-1}, x_m, ..., x_{n-1})$ dacă $m < n$,
  - $Z = (x_0, y_0, x_1, y_1, ..., x_{n-1}, y_{n-1}, y_n, ..., y_{m-1})$ dacă $n <= m$.

**Exerciţiul 2.3.4. (Problema lui Joseph)** Se consideră $n$ persoane aranjate în cerc. Începând cu o poziţie particulară $p_0$ se efectuează numărarea persoanelor şi, în mod brutal, se execută fiecare a $m$-a persoană; cercul se închide după eliminarea unei persoane. Execuţia se termină când rămâne o singură persoană; aceasta va rămâne în viaţă.

- Să se scrie un subprogram care crează o listă cu persoanele în ordinea în care au fost executate;
- Presupunând că se dorește salvarea unei anumite persoane, să se scrie un subprogram care să determine poziția $p_0$ astfel încât, dacă numărătoarea începe de pe această poziție, atunci persoana salvată să fie cea dorită.

## 2.4 Stive

### 2.4.1 Tipul de date abstract STIVA

#### 2.4.1.1 Descrierea obiectelor

*Stivele* sunt colecţii de date cu proprietatea că se cunoaşte „vechimea" fiecărui element, *i.e.*, se cunoaşte ordinea introducerii elementelor în colecţie. La orice moment, ultimul element introdus în colecţia de date este primul candidat la operaţiile de citire şi eliminare. Din acest motiv se mai numesc şi tipuri de date LIFO (Last In, First Out). Presupunem că elementele unei stive aparţin tipului abstract Elt.

#### 2.4.1.2 Operaţii

StivaVidă. Întoarce stiva vidă.

*Intrare:* – nimic;
*Ieşire:* – stiva fără nici un element.

EsteVidă. Testează dacă o stivă este vidă.

*Intrare:* – o stivă $S$;
*Ieşire:* – *true* dacă $S$ este stiva vidă,
– *false* în caz contrar.

Push. Scrie un element în stivă.

*Intrare:* – o stivă $S$ şi un element $e$ din Elt;
*Ieşire:* stiva $S$ la care s-a adăugat $e$. Elementul $e$ este ultimul introdus şi va fi primul candidat la operaţiile de citire şi eliminare.

Pop. Elimină ultimul element introdus în stivă.

*Intrare:* – o stivă $S$;
*Ieşire:* – stiva $S$ din care s-a eliminat ultimul element introdus dacă $S$ nu este vidă,
– *eroare* dacă $S$ este stiva vidă.

Top. Citeşte ultimul element introdus într-o stivă.
*Intrare:* – o stivă $S$;
*Ieşire:* – ultimul element introdus în $S$ dacă $S$ nu este stiva vidă,
– un mesaj de eroare dacă $S$ este stiva vidă.

**Observaţie.** Au fost folosite denumirile Push, Pop şi Top pentru că traducerea lor ar fi derutat cititorul. sfobs

**Observaţie.** Tipul abstract STIVA poate fi „implementat" şi de către tipul abstract LLIN în sensul că o stivă poate fi privită ca o listă liniară şi că operaţiile stivei pot fi exprimate cu ajutorul celor de la lista liniară:

STIVA.Push(S, e) = LLIN.Inserează(S, Lung(S), e)
STIVA.Top(S) = LLIN.Citeşte(S, Lung(S))
STIVA.Pop(S) = LLIN.Elimină(S, Lung(S))

Astfel, o stivă poate fi reprezentată printr-o secvenţă $(e_0, \ldots, e_{n-1})$ unde $e_{n-1}$ este elementul din vârful stivei (cu vechimea cea mai mică). Ca o consecinţă, obţinem următoarea interpretare pentru operaţii:

StivaVidă : $\mapsto ()$
Push : $((e_0, \ldots, e_{n-1}), e) \mapsto (e_0, \ldots, e_{n-1}, e)$
Pop : $(e_0, \ldots, e_{n-1}) \mapsto (e_0, \ldots, e_{n-2})$
Pop : $() \mapsto$ *eroare*
Top : $(e_0, \ldots, e_{n-1}) \mapsto e_{n-1}$
Top : $() \mapsto$ *eroare*

sfobs

## 2.4.2 Implementarea cu structuri dinamice simplu înlănţuite

### 2.4.2.1 Reprezentarea obiectelor

O stivă este o pereche formată dintr-o listă simplu înlănţuită, care conţine elementele memorate în stivă la un moment dat, şi o variabilă referinţă care indică vârful stivei, adică ultimul element introdus (figura 2.4).

### 2.4.2.2 Implementarea operaţiilor

StivaVidă. Constă în crearea listei simplu înlănţuite vide.

EsteVidă. Testează dacă vârful $S$ este egal cu NULL.

Push. (Inserează) Constă în adăugarea unui nod la începutul listei şi „mutarea" vârfului $S$ la nodul adăugat. Implementarea operaţiei necesită timpul $O(1)$.

```
procedure push(S, e)
    new(q)
    q->elt ← e
    q->succ ← S
    S ← q
end
```

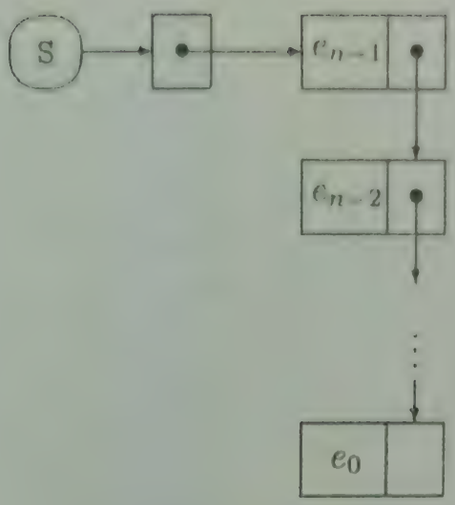



Figura 2.4: Stiva reprezentată ca listă înlănţuită

Pop. (Elimină) Constă în eliminarea nodului referit de pointerul $S$ şi „mutarea" vârfului la următorul nod. Ca şi implementarea operaţiei Push, necesită timpul $O(1)$.

```
procedure pop(S)
   if (S = NULL) then throw 'eroare'
   q ← S
   S ← S->succ
   delete(q)
end
```

Top. (Citeşte) Dacă stiva nu este vidă, furnizează informaţia memorată în nodul referit de vârful $S$.

```
function top(S)
   if (S = NULL) then throw 'eroare'
   return S->elt
end
```

## 2.4.3 Implementarea cu tablouri

### 2.4.3.1 Reprezentarea obiectelor

O stivă S este o pereche formată dintr-un tablou `S.tab`, care memorează elementele stivei, şi o variabilă indice (vârful) `S.varf` care indică poziţia ultimului element introdus în stivă (figura 2.5). Acesta poate fi utilizat şi pentru determinarea numărului de elemente din stivă (operaţiile standard nu cer această informaţie).

### 2.4.3.2 Implementarea operaţiilor

StivaVidă. Vârful, care memorează adresa ultimului element introdus în stivă, este poziţionat pe $-1$.

EsteVidă. Testează dacă vârful este $-1$.

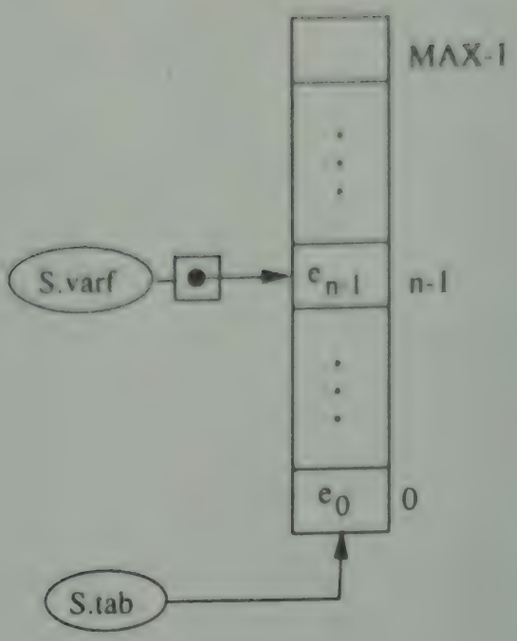



Figura 2.5: Reprezentarea stivei printr-un tablou

Push. Incrementează vârful şi memorează noul element în tablou la poziţia dată de noul vârf. Deoarece dimensiunea maximă a tabloului care memorează stiva este fixă, este necesar să se verifice dacă nu se depăşeşte această limită. Operaţia este realizată în timpul $O(1)$.

```
procedure push(S, e)
   if (S.varf = MAX-1) then throw 'memorie insuficienta'
   S.varf ← S.varf+1
   S.tab[S.varf] ← e
end
```

Pop. Dacă vârful este $\geq 0$, atunci acesta este decrementat. Evident, realizarea operaţiei se face în timpul $O(1)$.

```
procedure pop(S)
   if (S.varf < 0) then throw 'eroare'
   S.varf ← S.varf-1
end
```

Top. Dacă vârful este $\geq 0$, atunci întoarce elementul memorat în tablou la adresa dată de vârf.

```
function top(S)
   if (S.varf < 0) then throw 'eroare'
   return S.tab[S.varf]
end
```

### 2.4.4 Exerciţii

**Exerciţiul 2.4.1.** [Knu74] Se consideră o maşină compusă din trei memorii: o secţiune de intrare şi o secţiune de ieşire constând dintr-o secvenţă de $n$ celule fiecare (o celulă poate memora un număr întreg) şi o stivă cu memorie infinită. Controlul finit al maşinii este format dintr-un program construit cu următoarele instrucţiuni:

`Read`, introduce valoarea din prima celulă din secţiunea de intrare în stivă apoi deplasează conţinutul secţiunii de intrare astfel încât conţinutul din celula $k+1$ trece in celula $k$. Conţinutul din prima celulă se pierde, iar cel din ultima celulă devine nedefinit;
`Write`, deplasează conţinutul secţiunii de ieşire astfel încât conţinutul din celula $k$ trece în celula $k+1$. Conţinutul din ultima celulă se pierde apoi, extrage un element din stivă şi-l memorează în prima celulă din secţiunea de ieşire.

- Presupunând că secţiunea de intrare conţine $n$ valori distincte, să se determine numărul $a_n$ al permutărilor ce pot fi obţinute în secţiunea de ieşire.
- Să se scrie un program care enumera toate programele maşinilor ce realizează permutări.

**Exerciţiul 2.4.2.** Forma poloneză (notaţia posfixată) a unei expresii aritmetice se caracterizează prin faptul că operatorii apar în ordinea în care se execută operaţiile la evaluarea expresiei. De aceea, evaluarea unei expresii în forma poloneză se face parcurgând într-un singur sens expresia şi executând operaţiile ţinând seama de paritatea lor.
Definirea recursivă a expresiilor în forma poloneză este următoarea:

1. Orice operand (constantă sau variabilă) $E$ este în formă poloneză;
2. Dacă $E$ este o expresie de forma $E1 op E2$ , forma poloneză a expresiei este $E1'E2'op$, unde $E1'$ şi $E2'$ sunt respectiv f.p. ale expresiilor $E1$ şi $E2$
3. Dacă $E$ este o expresie de forma $(E1)$ forma poloneză a expresiei $E1$ este de asemenea f.p. a expresiei $E$

Exemple: $(9-5)+2$ are f.p. $9\ 5\ -\ 2\ +$, iar $9-(5+2)$ are f.p. $9\ 5\ 2\ +\ -$.
Observaţie. Expresiile în forma poloneză nu conţin paranteze.
Un algoritm simplu de evaluare a expresiilor în forma poloneză este următorul:

```
/* Expresia se afla intr-un tablou s cu elemente
simboluri de tip operand  sau operator binar */
i ← 0
while (nu s-a terminat sirul s)
   if (s[i] este operand)
      then depune s[i] in stiva
      else if (s[i] este operator)
              then extrage din stiva doua simboluri t1 si t2
                   executa operatia s[i] asupra lui t1 si t2
                   depune rezultatul in stiva
              else semnalizeaza eroare
   i ← i+1
```

Observaţie. Algoritmul presupune că expresiile conţin doar operatori binari.
Să se scrie un program care să rezolve problema evaluării unei expresii date în forma poloneză.

**Exerciţiul 2.4.3.** O altă formă fără paranteze a unei expresii aritmetice este forma prefixată.
Exemplu: $(9 - 5) + 2$ are forma prefixată $+ - 952$

- Să se formuleze o definiţie recursivă pentru forma prefixată a unei expresii aritmetice.
- Să se scrie un program de trecere a unei expresii din forma postfixată în forma prefixată.
- Să se scrie un program care să rezolve problema evaluării unei expresii date în forma prefixată.

## 2.5 Cozi

### 2.5.1 Tipul de date abstract COADA

#### 2.5.1.1 Descrierea obiectelor

Ca şi stivele, *cozile* sunt colecţii de date cu proprietatea că se cunoaşte vechimea fiecărui element. La orice moment, primul element introdus în colecţia de date este primul candidat la operaţiile de citire şi eliminare. Din acest motiv se mai numesc şi tipuri de date FIFO (First In, First Out). Presupunem că elementele unei stive aparţin tipului abstract Elt.

#### 2.5.1.2 Operaţii

CoadaVidă. Întoarce coada vidă.
*Intrare:* – nimic;
*Ieşire:* – coada fără nici un element.

EsteVidă. Testează dacă o coadă este vidă.
*Intrare:* – o coadă $C$;
*Ieşire:* – *true* dacă $C$ este coada vidă,
– *false* în caz contrar.

Inserează. Scrie un element în coadă.
*Intrare:* – o coadă $C$ şi un element $e$ din Elt;
*Ieşire:* – coada $C$ la care s-a adăugat $e$. Elementul $e$ este ultimul introdus şi va fi ultimul candidat la operaţiile de citire şi eliminare.

Elimină. Elimină elementul cel mai vechi din coadă.
*Intrare:* – o coadă $C$;
*Ieşire:* – coada $C$ din care s-a eliminat primul element introdus dacă $C$ nu este vidă,
– eroare dacă $C$ este coada vidă.

Citeşte. Citeşte elementul cel mai vechi din coadă.

*Intrare:* – o coadă $C$;
*Ieşire:* – elementul cel mai vechi din coada $C$ dacă $C$ nu este coada vidă.
– un mesaj de eroare dacă $C$ este coada vidă.

**Observaţie.** Ca şi în cazul stivei, tipul abstract COADA poate „implementat" şi de către tipul abstract LLIN în sensul că o coadă este privită ca o listă liniară şi că operaţiile cozii pot fi exprimate cu ajutorul celor de la lista liniară:

COADA.Inserează(C, e) = LLIN.Inserează(C, Lung(C), e)
COADA.Citeşte(C) = LLIN.Citeşte(C, 0)
COADA.Elimină(C) = LLIN.Elimină(S, 0)

Astfel, o coadă poate fi reprezentată printr-o secvenţă $(e_0, \ldots, e_{n-1})$, unde $e_0$ reprezintă capul cozii (elementul cu vechimea cea mai mare), iar $e_{n-1}$ este elementul de la sfârşitul cozii (cu vechimea cea mai mică). Utilizând reprezentarea prin secvenţe, operaţiile se rescriu astfel:

CoadaVidă : $\mapsto ()$
Inserează : $\langle (e_0, \ldots, e_{n-1}), e \rangle \mapsto (e_0, \ldots, e_{n-1}, e)$
Citeşte : $(e_0, \ldots, e_{n-1}) \mapsto e_0$
Citeşte : $() \mapsto$ *eroare*
Elimină : $(e_0, \ldots, e_{n-1}) \mapsto (e_1, \ldots, e_{n-1}$
Elimină : $() \mapsto$ *eroare*

sfobs

## 2.5.2 Implementarea cu structuri dinamice simplu înlănţuite

### 2.5.2.1 Reprezentarea obiectelor

O coadă este formată dintr-o listă simplu înlănţuită $C$, care conţine elementele memorate la un moment dat în coadă, o variabilă referinţă `C.prim` care face referire la nodul cu vechimea cea mai mare (candidatul la ştergere şi la interogare) şi o variabilă referinţă `C.ultim` care face referire la ultimul nod introdus în coadă (nodul cu vechimea cea mai mică) (figura 2.6).

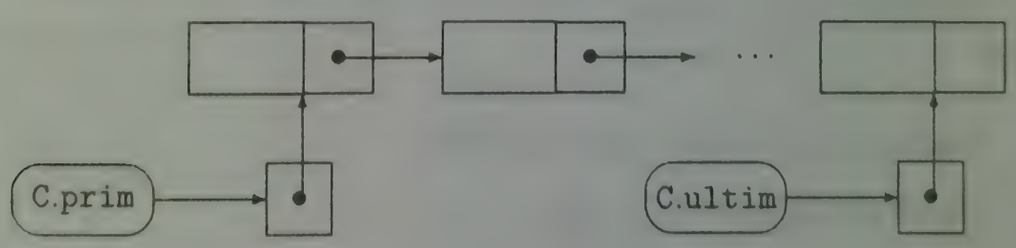



Figura 2.6: Coada reprezentată ca listă simplu înlănţuită

### 2.5.2.2 Implementarea operaţiilor

CoadaVidă. Creează lista simplu înlănţuită vidă.

EsteVidă. Testează dacă pointerul `C.prim` este egal cu `NULL`.

Inserează. Adaugă un nod după cel referit de `C.ultim` şi mută referinţa `C.ultim` la nodul nou introdus. În cazul când s-a adăugat la coada vidă, se actualizează şi referinţa `C.prim`.

```
procedure insereaza(C, e)
   new(q)
   q->elt ← e
   q->succ ← NULL
   if (C.ultim ≠ NULL)
      then C.ultim->succ ← q
      else C.prim ← q
   C.ultim ← q
end
```

Elimină. În cazul în care coada nu este vidă, elimină nodul referit de `C.prim` (cel cu vechimea cea mai mare) şi mută referinţa `C.prim` la nodul următor. Dacă lista avea un singur nod, atunci, după eliminare, va deveni vidă şi cei doi pointeri vor fi actualizaţi corespunzător.

```
procedure elimina(C)
   if (C.prim = NULL) then throw 'eroare'
   p ← C.prim
   C.prim ← C.prim->succ
   if (C.prim = NULL) then C.ultim ← NULL
   delete(p)
end
```

Citeşte. Dacă lista nu este vidă, furnizează informaţia din nodul cu vechimea cea mai mare (referit de `C.prim`).

```
function citeste(C)
   if (C.prim = NULL) then throw 'eroare'
   return C.prim->elt
end
```

### 2.5.3 Implementarea cu tablouri

#### 2.5.3.1 Reprezentarea obiectelor

Elementele unei cozi sunt memorate într-un tablou. Câmpul `C.prim`, care acum este adresă în tablou, referă cel mai vechi element în coadă, iar `C.ultim` cel mai nou element în coadă (figura 2.7). Pentru o mai bună utilizare a spaţiului de memorie vom conveni ca, atunci când cel mai nou element este memorat pe poziţia ultimă `MAX-1` din tablou, următorul element să fie memorat pe prima poziţie din tablou, dacă aceasta este liberă (figura 2.7.b). Aceasta conduce la utilizarea unei aritmetici modulare pentru câmpurile `C.prim` şi `C.ultim`. Mai rămâne de rezolvat problema

reprezentării cozii vide. Aceasta devine foarte simplă dacă vom considera o variabilă contor suplimentară C.nrElem care să memoreze numărul elementelor din coadă. În plus, pentru că valoarea lui C.ultim poate fi dedusă din valorile C.prim şi C.nrElem utilizarea cámpului C.ultim devine acum opţională.

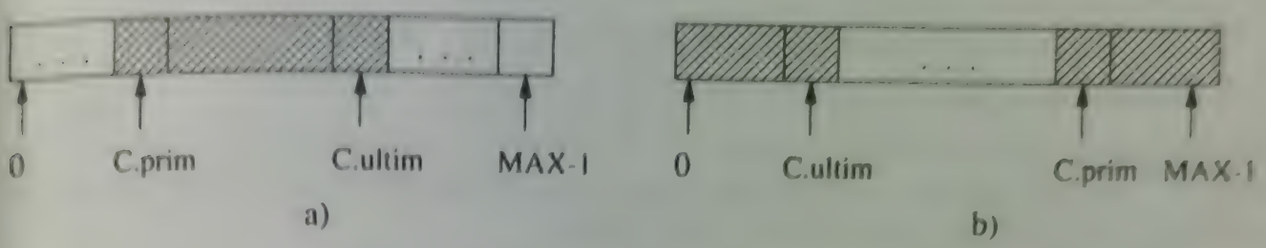



Figura 2.7: Reprezentarea cozii printr-un tablou

### 2.5.3.2 Implementarea operaţiilor

CoadaVidă. Variabila C.nrElem este pusă pe zero. Este convenabil ca C.ultim să fie făcut egal cu 0, iar C.prim să fie făcut egal $-1$.

EsteVidă. Testează dacă variabila C.nrElem este zero.

Inserează. Dacă mai este loc în tablou, C.nrElem < MAX, atunci incrementează C.ultim modulo MAX şi memorează noul element e la noua adresă dată de C.ultim. Variabila contor C.nrElem este incrementată.

```
procedure insereaza(C, e)
   if (C.nrElem = MAX) then throw 'memorie insuficienta'
   C.ultim ← (C.ultim + 1) % MAX
   C.tab[C.ultim] ← e
   C.nrElem ← C.nrElem + 1
end
```

Elimină. În cazul în care coada nu este vidă, incrementează pointerul C.prim modulo MAX şi decrementează variabila contor C.nrElem.

```
procedure elimina(S)
   if (C.nrElem = 0) then throw 'eroare'
   C.prim ← (C.prim + 1) % MAX
   C.nrElem ← C.nrElem - 1
end
```

Citeşte. Dacă coada nu este vidă, furnizează informaţia de la adresa din tablou dată de pointerul C.prim.

```
function citeste(C)
   if (C.nrElem = 0) then throw 'eroare'
   return C.tab[C.prim]
end
```

### 2.5.4 Exerciţii

**Exerciţiul 2.5.1.** O *coadă completă cu restricţie la intrare* este o listă liniară abstractă cu proprietatea că elementele se pot insera numai la unul din capete, iar ştergerile/citirile se pot efectua la oricare din capete.

- Să se implementeze această structură cu ajutorul listelor înlănţuite.
- Să se implementeze această structură cu ajutorul tablourilor.

**Exerciţiul 2.5.2.** O *coadă completă cu restricţie la ieşire* este o listă liniară abstractă cu proprietatea că elementele se pot insera la oricare din capete iar ştergerile/citirile se pot efectua numai la unul din capete.

- Să se implementeze această structură cu ajutorul listelor înlănţuite.
- Să se implementeze această structură cu ajutorul tablourilor.

**Exerciţiul 2.5.3.** O *coadă completă* (*deque*) este o listă liniară abstractă cu proprietatea că elementele se pot insera, şterge şi citi la ambele capete.

- Să se implementeze această structură cu ajutorul listelor înlănţuite.
- Să se implementeze această structură cu ajutorul tablourilor.

**Exerciţiul 2.5.4.** Se consideră un tablou unidimensional cu 10.000 de componente. Să se scrie subprograme care să realizeze:

- memorarea eficientă în tablou a două stive;
- memorarea în tablou a două cozi. Se pot memora două cozi la fel de eficient ca două stive? Dacă da, atunci să se arate cum, dacă nu să se argumenteze de ce.

## 2.6 Liste generalizate

### 2.6.1 Tipul de date abstract LGEN

#### 2.6.1.1 Descrierea obiectelor

O *listă generalizată* este o secvenţă finită $(e_0, \ldots, e_{n-1})$ în care elementele $e_i$ sunt fie atomice, fie liste generalizate. Elementele atomice aparţin unui tip dat Elt. Componentele unei liste generalizate nu trebuie să fie neapărat distincte; adică, putem avea $i \neq j$ şi $e_i = e_j$.

Listele generalizate sunt structuri definite recursiv. Elementele de tip listă ale unei liste generalizate se numesc subliste. În exemplul următor, lista generalizată $A$ este formată din elementul atomic $x$ şi sublistele $B$, $C$, $(a, x, c)$, $()$, unde $()$ semnifică lista vidă.

atom listă

$A = (x, B, C, (a, x, c), ())$

listă lista vidă

### 2.6.1.2 Operaţii

Operaţiile ListaVidă, EsteVidă, Copie şi Egal au definţii identice cu cele de la lista liniară.

Head. Întoarce primul element dintr-o listă generalizată.
*Intrare:* - o listă generalizată $L = (e_0, \ldots, e_{n-1})$;
*Ieşire:* - $e_0$ (primul element al listei $L$), dacă $L \neq ()$,
- *eroare*, dacă $L=()$.

De notat că Head($L$) poate fi atom sau sublistă.

Tail. Întoarce lista rezultată prin eliminarea dintr-o listă generalizată a primului element.
*Intrare:* - o listă generalizată $L = (e_0, e_1, \ldots, e_{n-1})$;
*Ieşire:* - $L' = (e_1, \ldots, e_{n-1})$ (lista rezultată din $L$ prin eliminarea din $L$ a primului element), dacă $L \neq ()$
- *eroare*, dacă $L=()$.

În continuare $L$ şi $L'$ partajează elementele $(e_0, \ldots, e_{n-1})$.

Lung. Întoarce lungimea unei listei generalizate date exprimată în număr de atomi.
*Intrare:* - o listă generalizată $L = (e_0, \ldots, e_{n-1})$;
*Ieşire:* - Lung($L$) = numărul $m$ al componentelor atomice ale listei:

$$\text{Lung}(L) = \begin{cases} 0, & \text{dacă } L \text{ este vidă;} \\ 1 + \text{Lung}(\text{Tail}(L)), & \text{dacă } e_o = \text{Head}(L) \text{ este atom;} \\ \text{Lung}(\text{Head}(L)) + \text{Lung}(\text{Tail}(L)), & \text{altfel.} \end{cases}$$

Pentru exemplul de la începutul secţiunii avem Lung($(a, x, c)$) = 1+Lung($(x, c)$) = 2+Lung($(c)$) = 3+Lung($()$) = 3+0 = 3 şi Lung($A$) = 1+Lung($(B, C, (a, x, c), ())$) = 1+Lung($B$)+Lung($(C, (a, x, c), ())$) = ... = 4+Lung($B$)+Lung($C$).

Parcurge. Traversarea listelor generalizate presupune vizitarea sistematică, fără repetare, a elementelor atomice ale listei.
*Intrare:* o listă generalizată $L = (e_0, \ldots, e_{n-1})$ şi o procedură Vizitează($atom$);
*Ieşire:* - lista $L$ cu componentele atomice prelucrate de procedura Vizitează.

*Observaţie.* Ieşirea operaţiei Parcurge poate fi definită recursiv, cu ajutorul operaţiilor Head şi Tail.

$$\text{Parcurge}(L) = \begin{cases} (), & \text{dacă } L \text{ este lista vidă;} \\ (\text{Vizitează}(e_0), \text{Parcurge}(\text{Tail}(L)), & \text{dacă } e_0 \text{ este atom;} \\ (\text{Parcurge}(\text{Head}(L)), \text{Parcurge}(\text{Tail}(L))), & \text{altfel.} \end{cases}$$

Inserează. Adaugă la început un element dat (atom sau listă generalizată) într-o listă generalizată dată.
*Intrare:* - o listă generalizată $L = (e_0, \ldots, e_{n-1})$ (posibil vidă) şi $x$ un atom sau o listă generalizată;
*Ieşire:* - $L = (x, e_0, \ldots, e_{n-1})$.

**Elimină.** Elimină un element dat (atom sau listă generalizată) dintr-o listă generalizată dată.

*Intrare:* – o listă generalizată $L = (e_0, \ldots, e_{n-1})$ (posibil vidă) şi $x$ un atom sau o listă generalizată;
*Ieşire:* – lista $L$ din care s-au eliminat toate aparitiile lui $x$.

## 2.6.2 Implementarea cu structuri dinamice înlănţuite

### 2.6.2.1 Reprezentarea obiectelor

O listă generalizată $L = (e_0, \ldots, e_{n-1})$ poate fi reprezentată printr-o structură dinamică înlănţuită. Notăm cu `L.prim` pointerul care face referire la primul element. Un element de listă generalizată va fi o structură statică `(tip, data, succ)`, unde câmpul `tip` conţine o valoare $tip \in \{Atom, Lista\}$. Dacă $tip = Atom$, atunci elementul va corespunde unui obiect atomic. În consecinţă, câmpul `data` va conţine valoarea atomului respectiv. Dacă $tip = Lista$, atunci elementul va corespunde unei subliste şi câmpul `data` va semnifica legătura la primul element al sublistei. Câmpul `succ` memorează informaţia de înlănţuire relativă la următorul element din listă.

Considerăm ca exemplu următoarele liste:
A=$(x, (y, z))$;
$B = (A, A, (), w)$.
O reprezentare a listelor generalizate $A$ şi $B$ este descrisă în figura 2.8.

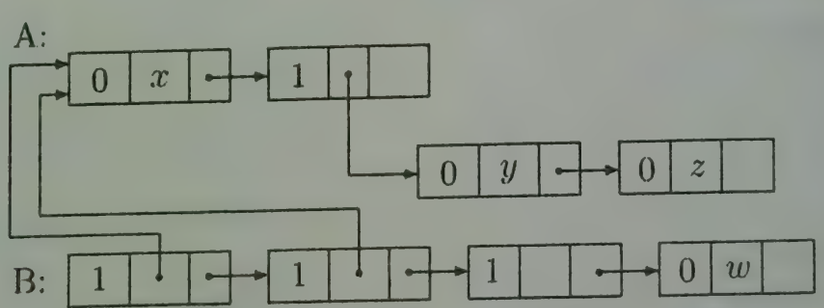



Figura 2.8: Reprezentarea listei generalizate cu structuri înlănţuite

Pentru a facilita referirea sublistelor, convenim să dăm nume sublistelor. Astfel, dacă $tip = Lista$, atunci *data* va fi adresa unei structuri statice ce conţine *(nume sublistă, pointer la primul element)*. Obţinem reprezentarea din figura 2.9.

### 2.6.2.2 Implementarea operaţiilor

Operaţiile ListaVidă, EsteVidă au implementări similare cu cele de la lista liniară.

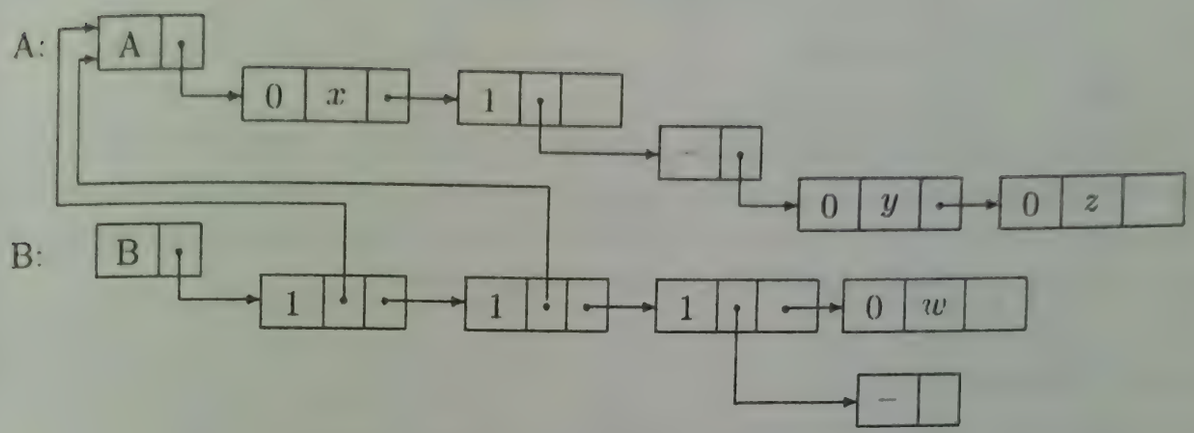



Figura 2.9:
Reprezentarea listei generalizate cu structuri înlănţuite unde sublistele au nume

Parcurge. Definiţia unei liste generalizate este recursivă. În consecinţă, operaţia de parcurgere a unei liste generalizate va fi realizată prin algoritmi recursivi.

```
procedure parcurge(L, viziteaza())
   p ← L.prim
   while (p ≠ NULL) do
      if ((tip(p) = ATOM)
         then viziteaza(p->elt)
         else parcurge(p->elt, viziteaza())
      p ← p->succ;
end
```

Se observă că dacă se face abstracţie de procesarea diferenţiată a elementelor în funcţie de tipul acestora (*Atom* sau *Lista*) se obţine algoritmul parcurgerii unei liste simplu înlănţuite. Procesarea sublistelor se face prin apeluri recursive.

Operaţia de parcurgere a unei liste generalizate are timpul de execuţie, pe cazul cel mai nefavorabil, $O(m)$, unde $m$ este numărul elementelor de tip atom.

Copie. Copierea unei liste generalizate $L$ are ca efect crearea unei dubluri a acesteia.

```
procedure Copie(L)
```

```
if (L = NULL)
   then return NULL
   else p ← new Elt
        if p->tip=Atom
           then p->data ← L->data
           else p->data ← Copie(L->data)
        p->succ  ← Copie(L->succ)
        return p
end
```

Implementarea celorlalte operaţii o lăsăm ca exerciţiu. Inserarea unui element într-o listă generalizată este asemănătoare cu cea de la liste liniare. Elementul care se inserează poate fi un atom sau o sublistă. Operaţia de eliminare a unui element trebuie să ţină seama de existenţa unor subliste comune.

### 2.6.3 Exerciţii

**Exerciţiul 2.6.1.** Să se definească recursiv operaţia Elimină.

**Exerciţiul 2.6.2.** Se defineşte *Liniarizare*$(L)$ ca fiind lista liniară $(a_0, \ldots, a_{k-1}, a_k, \ldots, a_{m-1})$ a atomilor listei generalizate $L$, scrişi în ordinea dată de operaţia Parcurge. Să se definească cu ajutorul noţiunii *Liniarizare*$(L)$ următoarele operaţii:
Poz. Întoarce poziţia unui atom dat dintr-o listă generalizată dată. Dacă atomul nu se află în listă, întoarce $-1$.
CiteşteAtom. Întoarce atomul de la o poziţie dată dintr-o listă generalizată dată. Poziţia este dată de numărul de ordine rezultat din parcurgerea listei generalizate (numărul de ordine în lista *Liniarizare*$(L)$). Dacă atomul nu se află în listă, întoarce *'eroare'*.
Inserează̆Atom. Adaugă un atom dat într-o listă generalizată dată la o poziţie dată.
Inserează̆Sublistă. Adaugă o sublistă dată într-o listă generalizată dată la o poziţie dată.

**Exerciţiul 2.6.3.** Să se definească recursiv operaţiile Poz, CiteşteAtom, Inserează̆Atom şi Inserează̆Sublistă.

**Exerciţiul 2.6.4.** Să se scrie subprograme care implementează operaţiile Egal, Lung, Poz, CiteşteAtom, Inserează̆Atom şi Inserează̆Sublistă.

**Exerciţiul 2.6.5.** Două liste generalizate se numesc echivalente dacă mulţimile atomilor sunt egale, iar sublistele sunt echivalente două câte două (nu contează ordinea). Mai precis, avem:

1. doi atomi sunt echivalenţi ddacă sunt egali;
2. două liste $L = \{e_0, \ldots, e_{n-1}\}$ şi $L' = \{e'_0, \ldots, e'_{n-1}\}$ sunt echivelente ddacă $n = n'$ şi pentru orice $i \in \{0, \ldots, n-1\}$ există un unic $j$ a.î. $e_i$ este echivalent cu $e'_j$.

De exemplu, $(A, (B, C), (D)) \equiv ((D), A, (C, B))$.

Să se scrie un subprogram care să determine dacă două liste generalizate sunt echivalente.

# 2.7 Arbori binari

## 2.7.1 Tipul de date abstract ABIN

### 2.7.1.1 Descrierea obiectelor

Mulţimea *arborilor binari* peste Elt este definită recursiv astfel:

- arborele vid [] este un arbore binar,
- arborele $t$ format dintr-un nod distinct, numit *rădăcină*, ce memorează un element $e$ din Elt, şi din doi arbori binari $t_1$ şi $t_2$, numiţi *subarborele stâng* şi respectiv *subarborele drept* (ai rădăcinii), este un arbore binar; notăm acest arbore prin $[e](t_1, t_2)$.

Convenim ca arborele cu un singur nod $[e]([\,],[\,])$ să mai fie notat şi prin notaţia mai simplă $[e]$. Fie $v$ un nod într-un arbore binar $t$ şi $v_1$ şi $v_2$ rădăcinile subarborilor stâng şi respectiv drept ai nodului $v$. Atunci spunem că $v$ este *nod tată* pentru $v_1$ şi $v_2$, $v_1$ este *fiul stâng* al lui $v$, iar $v_2$ este *fiul drept* al lui $v$. Un nod pentru care ambii subarbori sunt vizi este numit *frunză*. Totalitatea nodurilor frunză formează *frontiera* arborelui. Un *drum* într-un arbore este o succesiune de noduri $v_1, \ldots, v_n$ cu proprietatea că $v_i$ este tatăl lui $v_{i+1}$ pentru $i = 1, \ldots, n-1$. *Lungimea* unui drum $e_1, \ldots, e_n$ este $n-1$. *Dimensiunea* unui arbore $t$ este egală cu numărul de noduri din $t$. *Înălţimea* unui arbore $t$ este lungimea celui mai lung drum de la rădăcina lui $t$ la un nod frunză. Prin definiţie, înălţimea arborelui vid este $-1$.

**Exemplu.** Fie $a, b, c, d, e, f, g, h, i$ nouă elemente în Elt. Aplicând de mai multe ori partea a doua a definiţiei obţinem succesiv arborii binari $[a]([\,],[d])$, $[b]([f],[i])$, $[g]([h],[\,])$, $[e]([b]([f],[i]),[g]([h],[\,]))$, $[c]([a]([\,],[d]),[e]([b]([f],[i]),[g]([h],[\,])))$. Ultimul arbore binar este reprezentat în figura 2.10. De exemplu, $e$ este tatăl nodurilor $b$ şi $g$, $b$ este fiul stâng al lui $e$ iar $g$ este fiul drept al lui $e$. Înălţimea arborelui este 3, iar dimensiunea sa este 9. sfex

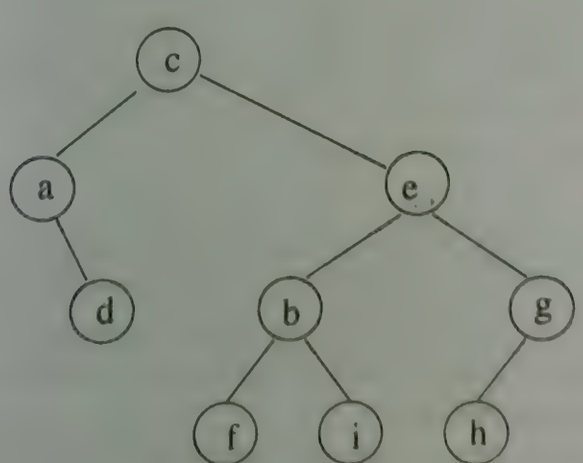



Figura 2.10: Exemplu de arbore binar

#### 2.7.1.2 Operaţii

ArbBinVid.

*Intrare:* - nimic;
*Ieşire:* - arborele binar vid.

EsteVid.

*Intrare:* - un arbore binar $t$;
*Ieşire:* - *true* dacă $t$ este vid,
- *false* în caz contrar.

Dimensiune.

*Intrare:* - un arbore binar $t$;
*Ieşire:* - dimensiunea lui $t$.

Înălţime.

*Intrare:* - un arbore binar $t$;
*Ieşire:* - înălţimea arborelui $t$.

Copie.

*Intrare:* - un arbore binar $t$;
*Ieşire:* - o copie distinctă a lui $t$.

Egal.

*Intrare:* - doi arbori binari $t_1$ şi $t_2$;
*Ieşire:* - *true* dacă cei doi arbori sunt egali, *i.e.*, elementul memorat în rădăcina lui $t_1$ este egal cu cel memorat de rădăcina lui $t_2$ şi subarborele stâng al lui $t_1$ este egal cu subarborele stâng al lui $t_2$ şi subarborele drept al lui $t_1$ este egal cu subarborele drept al lui $t_2$. Doi arbori vizi sunt egali prin definiţie. Formal, $t_1 = t_2$ dacă şi numai dacă:

- $t_1 = [\,] = t_2$ sau
- $t_1 = [a](t_1', t_1'')$, $t_2 = [a](t_2', t_2'')$ şi $t_1' = t_2'$, $t_1'' = t_2''$.

- *false* în caz contrar.

ParcurgePreordine.

*Intrare:* - un arbore binar $t$ şi o procedură Vizitează(adr_nod);
*Ieşire:* - arborele $t$, dar în care nodurile au fost procesate în ordinea dată de parcurgerea preordine a lui $t$. Parcurgerea *preordine* a unui arbore $t$ este obţinută aplicând următoarea secvenţă de operaţii:

- vizitează rădăcina;
- parcurge preordine subarborele stâng;
- parcurge preordine subarborele drept.

De exemplu, parcurgerea preordine a arborelui din figura 2.10 produce lista: $c, a, d, e, b, f, i, g, h$.

ParcurgeInordine.

*Intrare:* – un arbore binar $t$ şi o procedură Vizitează(adr_nod);
*Ieşire:* – arborele $t$, dar în care nodurile au fost procesate în ordinea dată de parcurgerea inordine a lui $t$. Parcurgerea *inordine* a unui arbore $t$ este obţinută aplicând următoarea secvenţă de operaţii:

- parcurge inordine subarborele stâng;
- vizitează rădăcina;
- parcurge inordine subarborele drept.

De exemplu, parcurgerea inordine a arborelui din figura 2.10 produce lista: $a, d, c, f, b, i, e, h, g$.

ParcurgePostordine.

*Intrare:* – un arbore binar $t$ şi o procedură Vizitează(adr_nod);
*Ieşire:* – arborele $t$, dar în care nodurile au fost procesate în ordinea dată de parcurgerea postordine a lui $t$. Parcurgerea *postordine* a unui arbore $t$ este obţinută aplicând următoarea secvenţă de operaţii:

- parcurge postordine subarborele stâng;
- parcurge postordine subarborele drept;
- vizitează rădăcina.

De exemplu, parcurgerea postordine a arborelui din figura 2.10 produce lista $d, a, f, i, b, h, g, e, c$.

ParcurgeBFS.

*Intrare:* – un arbore binar $t$ şi o procedură Vizitează(adr_nod);
*Ieşire:* – arborele $t$, dar în care nodurile au fost procesate în ordinea dată de parcurgerea BFS a lui $t$. *Parcurgerea BFS* (*Breadth-First-Search*) a unui arbore $t$ constă în: vizitarea rădăcinii, apoi a fiilor rădăcinii (nodurile aflate la distanţa 1 faţă de rădăcină), apoi a nodurilor aflate la distanţa 2 faţă de rădăcină ş.a.m.d. De exemplu, parcurgerea BFS a arborelui din figura 2.10 produce lista: $c, a, e, d, b, g, f, i, h$.

## 2.7.2 Implementarea cu structuri dinamice înlănţuite

### 2.7.2.1 Descrierea obiectelor

Fiecare nod al arborelui binar este reprezentat printr-o structură care, în forma sa cea mai simplă, are trei câmpuri: un câmp `elt` pentru memorarea informaţiei din nod, un câmp `stg` pentru memorarea adresei rădăcinii subarborelui stâng şi un câmp `drp` pentru memorarea adresei rădăcinii subarborelui drept. Accesul la întreaga structură de date este realizat prin intermediul rădăcinii. Figura 2.11 descrie reprezentarea dinamică a arborelui binar din figura 2.10.

### 2.7.2.2 Implementarea operaţiilor

ArbBinVid. Constă în crearea structurii vide (`t` ← `NUll`).

EsteVid. Testează dacă rădăcina $t$ face referire la vreun nod.

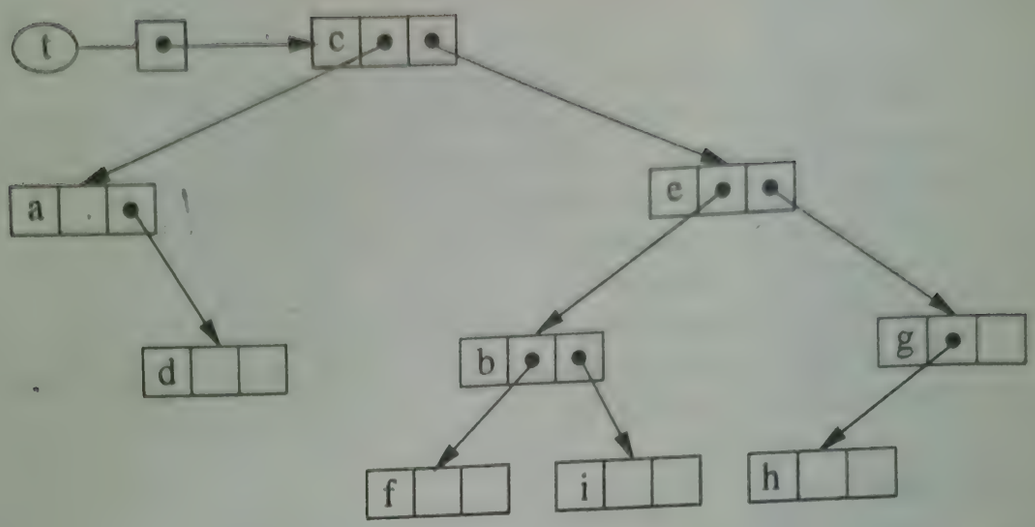



Figura 2.11: Reprezentarea arborelui binar cu structuri dinamice înlănţuite

Dimensiune. Descrierea funcţiei care calculează dimensiunea unui arbore binar are un caracter recursiv: dacă arborele este vid, atunci întoarce 0; dacă arborele nu este vid, atunci întoarce suma dintre dimensiunea subarborelui stâng şi dimensiunea subarborelui drept la care se adaugă 1 (rădăcina). Timpul de execuţie este $O(n)$, unde $n$ este chiar dimensiunea arborelui.

```
function dimens(t)
   if (t = NULL)
      then return 0
      else d1 ← dimens(t->stg)
           d2 ← dimens(t->drp)
           return d1 + d2 + 1
end
```

Înălţime. Funcţia care calculează înălţimea are o descriere asemănătoare celei care calculează dimensiunea. Timpul de execuţie este $O(n)$.

```
function inalt(t)
   if (t = NULL)
      then return -1
      else h1 ← inalt(t->stg)
           h2 ← inalt(t->drp)
           return max(h1, h2) + 1
end
```

Copie. Şi această funcţie are o descriere recursivă: se realizează o copie a rădăcinii după care se realizează prin apeluri recursive copii ale subarborilor rădăcinii. Lanţul apelurilor recursive se termină atunci când este întâlnit subarborele vid. Timpul de execuţie pentru cazul cel mai nefavorabil este $O(n)$.

```
function copie(t)
   if (t = NULL)
```

```
        then return NULL
        else new(t_copie)
             t_copie->elt ← t->elt
             t_copie->stg ← copie(t->stg)
             t_copie->drp ← copie(t->drp)
             return t_copie
end
```

Egal. Deoarece descrierea funcţiei egal este foarte simplă, omitem scrierea ei.

ParcurgePreordine. Din nou, cea mai simplă descriere este cea recursivă:

```
procedure preordine(t, viziteaza())
   if (t ≠ NULL)
      then viziteaza(t)
           preordine(t->stg, viziteaza())
           preordine(t->drp, viziteaza())
end
```

ParcurgeInordine. Este similară parcurgerii preordine.

ParcurgePostordine. Este similară parcurgerii preordine.

ParcurgeBFS. Pentru implementarea acestei operaţii se utilizează o coadă $C$ cu următoarele proprietăţi:

1. iniţial coada conţine numai rădăcina subarborelui;
2. la momentul curent se vizitează nodul extras din $C$;
3. atunci când un nod este vizitat, fiii acestuia sunt adăugaţi în $C$.

Timpul de execuţie pentru cazul cel mai nefavorabil este $O(n)$.

```
procedure parcurgeBFS(t, viziteaza())
   if (t ≠ NULL)
      then C ← coadaVida()
           insereaza(C,t)
           while not esteVida(C) do
              v ← citeste(C)
              elimina(C)
              viziteaza(v)
              if (t->stg ≠ NULL) then insereaza(C, t->stg)
              if (t->drp ≠ NULL) then insereaza(C, t->drp)
end
```

### 2.7.3 Implementarea cu tablouri

Nodurile arborelui binar sunt memorate într-un tablou de structuri. Valorile câmpurilor de legătură nu mai sunt adrese de variabil,e ci indici în tablou. O reprezentare statică a arborelui din figura 2.10 este dată în figura 2.12. Valoarea $-1$ joacă rolul pointerului NULL. Descrierile operaţiilor sunt asemănătoare cu cele de la implementarea dinamică. Prezentăm, ca exemplu, parcurgerea inordine:

```
procedure inordine(t, i, viziteaza())
   if (i ≠ -1)
      then inordine(t, t.tab[i].stg, viziteaza())
           viziteaza(t, i)
           inordine(t, t.tab[i].drp, viziteaza())
end
```

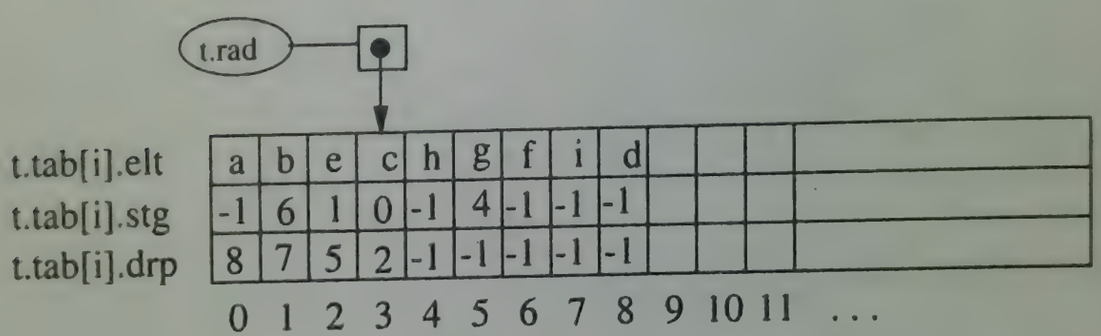



Figura 2.12: Reprezentarea cu tablouri a arborelui binar

### 2.7.4 Exerciţii

**Exerciţiul 2.7.1.** Să se scrie proceduri care implementează operaţiile pentru arborii binari reprezentaţi cu tablouri.

**Exerciţiul 2.7.2.** Asociem vârfurilor unui arbore binar adrese simbolice, ce sunt secvenţe de 0 şi 1 construite astfel: rădăcina are adresa simbolică 1; dacă vârful $v$ are adresa simbolică $\rho$, atunci fiul aflat la stânga lui $v$ are adresa simbolică $\rho 0$, iar fiul aflat la dreapta are adresa $\rho 1$.

a) Să se scrie un program care, pentru un arbore dat, enumeră adresele simbolice ale tuturor vârfurilor în preordine.

b) Să se scrie un program care, având la intrare adresele simbolice ale tuturor vârfurilor, construieşte o listă înlănţuită care reprezintă arborele.

**Exerciţiul 2.7.3.** Să se scrie un program care, având la intrare listele liniare ale nodurilor unui arbore binar în preordine, respectiv în inordine, construieşte lista înlănţuită care reprezintă arborele. Mai este posibil acelaşi lucru dacă se consideră la intrare oricare altă pereche de liste (preordine şi postordine sau inordine şi postordine)?

**Exercițiul 2.7.4.** Fie $t$ un arbore binar cu $n$ noduri, $v_1, \ldots, v_n$ lista nodurilor în preordine și $v_{p_1}, \ldots, v_{p_n}$ lista nodurilor în inordine. Să se arate că permutarea $p_1, \ldots, p_n$ se poate obține cu o mașină de la exercițiul 2.4.1 și reciproc, orice permutare obținută cu o mașină de la exercițiul 2.4.1 corespunde unui arbore binar.

**Exercițiul 2.7.5.** Presupunem că Elt este total ordonat. Peste arborii binari se definește relația $\prec$ astfel: $t_1 \prec t_2$ dacă și numai dacă:

$t_1 = [\,]$ sau
$t_1 = [a_1](t_1', t_1''),\ t_2 = [a_2](t_2', t_2'')$ și
$a_1 < a_2$ sau
$a_1 = a_2$ și $t_1' \prec t_2'$ sau
$a_1 = a_2$ și $t_1' = t_2'$ și $t_1'' \prec t_2''$.

a) Să se arate că, pentru orice doi arbori binari $t_1$ și $t_2$, are loc $t_1 \prec t_2$, $t_1 = t_2$ sau $t_2 \prec t_1$.

b) Să se scrie un program care, având la intrare doi arbori $t_1$ și $t_2$, decide dacă $t_1 \prec t_2$ sau $t_2 \prec t_1$ sau $t_1 = t_2$.

**Exercițiul 2.7.6.** (*Arbori binari însăilați la dreapta.*) Presupunem că structura unui nod cuprinde un câmp suplimentar `tdrp` cu următoarea semnificație:

- *v->tdrp =false* semnifică faptul că $v$ nu are fiu dreapta, iar *v->drp* va conține adresa succesorului lui $v$ din lista inordine. Un astfel de pointer se numește *însăilare*;
- *v->tdrp =true* semnifică faptul că $v$ are fiu dreapta.

În plus, se presupune că primul nod este succesorul-inordine al ultimului nod din lista inordine. Un astfel de arbore se numește *însăilat la dreapta.*

a) Să se scrie un subprogram `sucInord(p, t)` care determină succesorul-inordine al unui nod arbitrar $p$ în arborele $t$. Procedura va utiliza $O(1)$ spațiu suplimentar. Care este timpul de execuție a algoritmului descris de subprogram pentru cazul cel mai nefavorabil?

b) Este posibil să se scrie o procedură `sucInord` pentru arborii binari neînsăilați la dreapta? Dacă da, să se arate cum, iar dacă nu să se spună de ce.

c) Să se descrie o procedură de parcurgere a arborilor binari însăilați la dreapta, utilizând procedura `sucInord`. Să se arate că algoritmul de parcurgere descris necesită $O(n)$ timp ($n$ este numărul de noduri din arbore).

**Exercițiul 2.7.7.** Se consideră următoarea modificare a structurii de arbore binar:

- zona de legături a fiecărui nod conține două câmpuri suplimentare: `tstg` și `tdrp` cu semnificațiile:
  - *v->tstg =true*, dacă $v$ are subarbore stânga,
  - *v->tstg =false*, dacă $v$ nu are subarbore stânga și în acest caz *v->stg* este adresa predecesorului lui $v$ în inordine,
  - *v->tdrp =true*, dacă $v$ are subarbore dreapta,
  - *v->tdrp =false*, dacă $v$ nu are subarbore dreapta și în acest caz *v->drp* este adresa succesorului lui $v$ în inordine.

Un asemenea arbore se numește *însăilat.*

b) Să se scrie o procedură de parcurgere în inordine a arborilor însăilați.

(ii) Să se scrie o procedură care transformă un arbore binar obişnuit într-un arbore însăilat.

**Exerciţiul 2.7.8.** Să se scrie o procedură care să numere nodurile de pe frontiera unui arbore binar.

**Exerciţiul 2.7.9.** Se consideră arbori binari care au ca informaţie în noduri numere întregi. *Lungimea externă ponderată* a arborelui $t$ este

$$\sum\{v\text{->}inf * lung(v) \mid v \text{ este nod pe frontiera lui } t\},$$

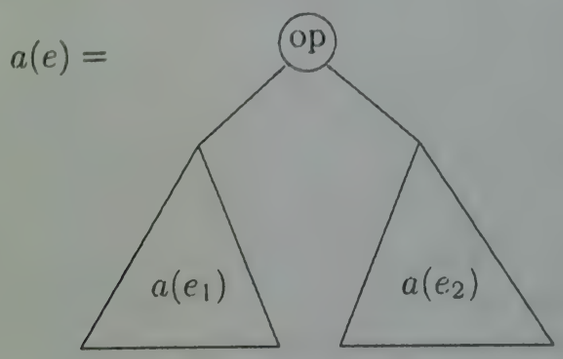
unde $lung(v)$ este lungimea drumului de la rădăcină la nodul $v$. Să se scrie un subprogram care determină lungimea externă a unui astfel de arbore.

**Exerciţiul 2.7.10.** (*Reprezentarea expresiilor aritmetice*) Considerăm expresii formate numai din numere întregi şi operatorii binari $+$, $-$ , $/$,$*$, $\%$. Unei expresii $e$ i se poate asocia un arbore binar $a(e)$ astfel:

- dacă $e$ este un număr întreg atunci $a(e)$ este format dintr-un singur nod $[e]$:

$$a(e) = \textcircled{e}$$

- dacă $e = e_1 \,\mathrm{op}\, e_2$ atunci $a(e)$ este arborele $[\mathrm{op}](a(e_1), a(e_2))$:




Rădăcina lui $a(e)$ are eticheta „op", subarborele din stânga rădăcinii este $a(e_1)$ iar subarborele din dreapta rădăcinii este $a(e_2)$.

Să se scrie un subprogram care are la intrare o listă liniară ce reprezintă o expresie şi oferă la ieşire arborele binar asociat expresiei. Arborele asociat este unic? Să se justifice cum rezolvă programul problema unicităţii.

**Exerciţiul 2.7.11.** Să se arate că notaţia postfixată se obţine prin parcurgerea postordine.

**Exerciţiul 2.7.12.** Să se arate că notaţia prefixată se obţine prin parcurgerea preordine.

**Exerciţiul 2.7.13.** Fie $T$ un arbore binar reprezentat prin structuri dinamice înlănţuite. Să se scrie un subprogram care să numere frunzele lui $T$ care au frate.

**Exerciţiul 2.7.14.** Fie $T$ un arbore binar reprezentat prin structuri dinamice înlănţuite. Să se scrie un subprogram care să numere nodurile de pe frontiera lui $T$ care au nivelul egal cu înălţimea arborelui.

# 2.8 Coada cu priorităţi

## 2.8.1 Tipul de date abstract coadă cu priorităţi

### 2.8.1.1 Obiectele

O *coadă cu priorităţi* este o structură de date în care elementele sunt numite *atomi*, iar fiecare atom conţine un câmp-cheie care ia valori dintr-o mulţime total ordonată. Valoarea acestui câmp-cheie se numeşte *prioritate*. Operaţiile de citire/eliminare se referă întotdeauna la atomul cu prioritatea cea mai mare. Interpretarea noţiunii de *prioritate* poate diferi de la caz la caz. Există situaţii când atomii cei mai prioritari sunt cei cu cheile valorilor mai mici şi există situaţii când atomii cei mai prioritari sunt cei cu cheile valorilor mai mari. Noi considerăm aici ultimul caz.

### 2.8.1.2 Operaţii

CoadaVidă.

*Intrare:* – nimic;
*Ieşire:* – coada cu priorităţi vidă.

Elimină.

*Intrare:* – o coadă cu priorităţi $Q$;
*Ieşire:* – $Q$ din care s-a eliminat atomul cu cheia cea mai mare (dacă există).

Inserează.

*Intrare:* – o coadă cu priorităţi $Q$ şi un atom $a$;
*Ieşire:* – $Q$ la care s-a adăugat $a$.

Citeşte.

*Intrare:* – o coadă cu priorităţi $Q$;
*Ieşire:* – atomul cu cheia cea mai mare din coada $Q$.

## 2.8.2 Implementarea cu max-*heap*-uri

### 2.8.2.1 Descrierea max-*heap*-urilor

**Definiţia 2.1.** *Un max-heap este un arbore binar complet cu proprietăţile:*

1. *informaţiile din noduri sunt valori dintr-o mulţime total ordonată numite* chei*;*
2. *pentru orice nod intern* v*, cheia memorată în* v *este mai mare sau egală cu cheia oricăruia dintre fii.*

În continuare vom arăta cum max-*heap*-urile sunt reprezentate prin tablouri 1-dimensionale.

**Definiţia 2.2.** *Pentru un $n \in N^*$,* arborele binar complet *ataşat lui $n$ este definit prin*

$$H_n = (\{0, \ldots, n-1\}, A)$$

*unde $A$ este mulţimea de arce $\{(\left[\frac{i-1}{2}\right], i) \mid i = 1, \ldots, n-1\}$.*

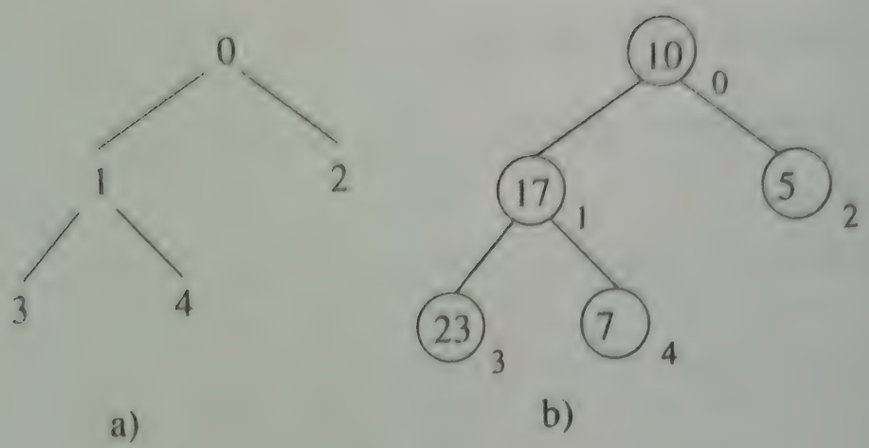



Figura 2.13: Arbore binar complet

**Exemplu.** Pentru $n = 5$, arborele $H_5$ este reprezentat în figura 2.13a. Mulţimea arcelor este $A = \{(0,1),(0,2),(1,3),(1,4)\}$. sfex

**Lema 2.1.** *Au loc următoarele proprietăţi ale arborelui* $H_n$*:*

1. *Vârful* $i$ *are succesorii* $2i+1$ *şi* $2i+2$.
2. *Vârful* $i$ *are ca vârf-tată pe* $\left[\frac{i-1}{2}\right]$, *dacă* $i \geq 1$.
3. *Arborele are* $[\log_2 n] + 1$ *nivele.*
4. *Pe nivelul* $k$ *se găsesc vârfurile* $2^k - 1, 2^k, \ldots, \min\{2^{k+1} - 2, n-1\}$.[1]

**Definiţia 2.3.** *Fie* $a = (a_0, \ldots, a_{n-1})$. *Prin* $H_n(a)$ *notăm arborele obţinut din* $H_n$ *prin etichetarea vârfurilor* $i$ *cu elementele* $a_i$ *în mod corespunzător.*

**Exemplu.** Dacă $a = (10, 17, 5, 23, 7)$, atunci arborele $H_5(a)$ este reprezentat în figura 2.13b. sfex

**Definiţia 2.4.** a) *Tabloul* $(\mathtt{a[i]} \mid i = 0, \ldots, n-1)$ *are proprietatea* MAX-HEAP, *dacă* $(\forall k)(1 \leq k < n \Rightarrow a\left[\frac{k-1}{2}\right] \geq a[k])$. *Notăm această proprietate prin* MAX-HEAP(a).

b) *Tabloul* a *are proprietatea* MAX-HEAP *începând cu poziţia* $\ell$ *dacă* $(\forall k)(\ell \leq \frac{k-1}{2} < k < n \Rightarrow a\left[\frac{k-1}{2}\right] \geq a[k])$. *Notăm această proprietate cu* MAX-HEAP(a, $\ell$).

Vom da câteva proprietăţi ale predicatelor MAX-HEAP(a) şi MAX-HEAP(a, $\ell$).

**Lema 2.2.** *1. Tabloul* $(\mathtt{a[i]} \mid i = 0, \ldots, n-1)$ *are proprietatea* MAX-HEAP *(adică* MAX-HEAP(a) =*true) dacă şi numai dacă pentru orice vârf* $a[i]$ *din* $H_n(a)$, $a[i]$ *este mai mare decât sau egal cu orice succesor* $a[j]$ *al său în* $H_n(a)$.

*2. Pentru orice tablou* $(\mathtt{a[i]} \mid i = 0, \ldots, n-1)$ *are loc* MAX-HEAP(a, $\left[\frac{n}{2}\right]$).

*3. Dacă* MAX-HEAP(a, $\ell$), *atunci pentru orice* $j > \ell$ *are loc* MAX-HEAP(a, $j$).

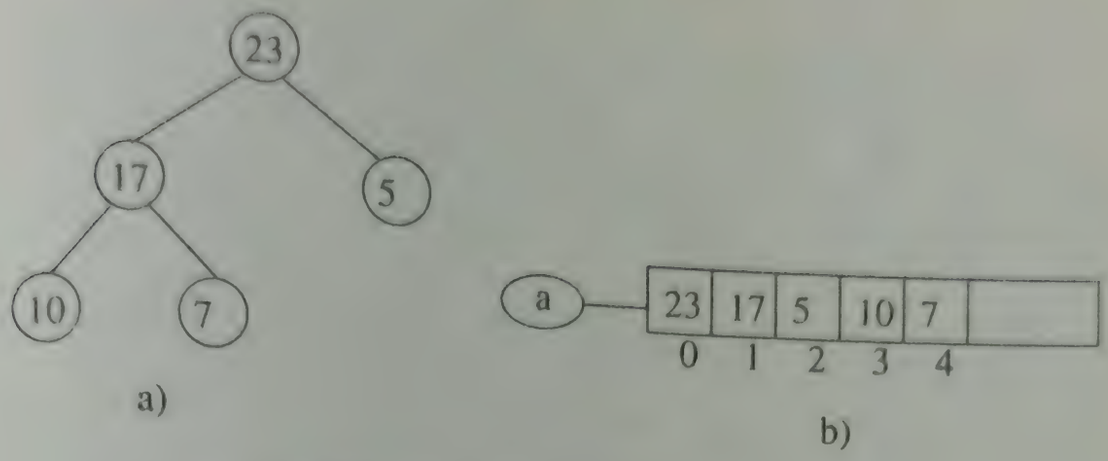
*4. Dacă* MAX-HEAP(a) *atunci* $a[0]$ *este elementul maxim din tablou.*

*5. Dacă* MAX-HEAP(a) *atunci* $H_n(a)$ *este un max-heap.*

În figura 2.14 este prezentat un exemplu de max-heap însoţit de tabloul cu reprezentarea sa.

**Teorema 2.1.** *Fie* $a = (a_0, \ldots, a_{n-1})$. *Atunci* $H_n(a)$ *este* max-heap *dacă şi numai dacă* MAX-HEAP(a).

[1] Aici nivelurile sunt numerotate începând cu 0; rădăcina este pe nivelul 0.




a) b)

Figura 2.14: Exemplu de max-*heap*

#### 2.8.2.2 Implementarea operaţiilor

CoadaVidă. Este reprezentată de tabloul vid.

Elimină. Atomul cu cea mai mare cheie se află în rădăcina max-*heap*-ului (prima poziţie în tablou). Ştergerea acestuia lasă un loc liber în rădăcină în care copiem atomul de pe ultima poziţie din tablou. Dimensiunea max-*heap*-ului este decrementată cu 1. Refacerea proprietăţii MAX-HEAP se realizează prin parcurgerea unui drum de la rădăcină spre frontieră conform următorului algoritm:

1. Presupunem că vârful curent este j. Iniţial avem $j = 0$.
2. Dacă $2 * j + 1 \leq n - 1$, atunci determină vârful cu valoarea maximă dintre fiii lui $j$; fie acesta $k$. Altfel, refacerea proprietăţii este terminată.
3. Dacă a[j] $\geq$ a[k], atunci refacerea proprietăţii este terminată.
4. Dacă a[j] $<$ a[k], atunci
   (a) Interschimbă a[j] cu a[k].
   (b) Repetă pasul 2 cu vârful curent $j \leftarrow k$.

Descrierea completă a algoritmului de eliminare este:

```
procedure elimina(a, n)
   a[0] ← a[n-1]
   n ← n-1
   j ← 0
   esteHeap ← false
   while ((2*j+1 ≤ n-1) and not esteHeap)
      k ← 2*j+1
      if ((k < n-1) and (a[k] < a[k+1])) then k ← k+1
      if (a[j] < a[k])
         then swap(a[j], a[k])
         else esteHeap ← true
      j ← k
   esteHeap ← true
end
```

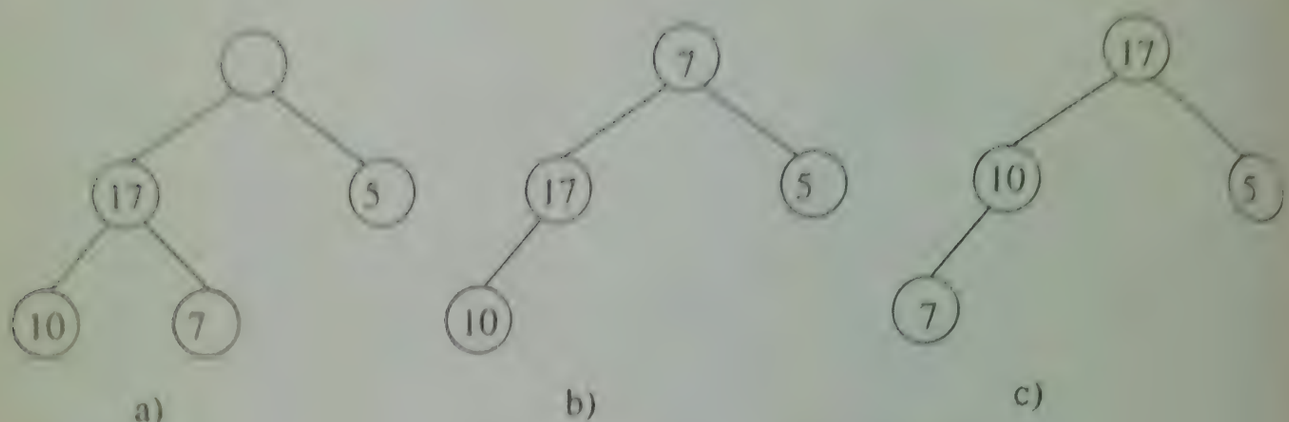



Figura 2.15: Eliminarea din max-*heap*-ul din figura 2.14

Timpul de execuţie în cazul cel mai nefavorabil este $O(\log_2 n)$. În figura 2.15 este arătat modul în care este eliminat atomul cu cheia cea mai mare din max-*heap*-ul din figura 2.14.

Inserează. Presupunând că max-*heap*-ul are $n$ elemente, se memorează noul atom pe poziţia $n$ şi se incrementează dimensiunea max-*heap*-ului. Apoi se reface proprietatea MAX-HEAP parcurgând un drum de la nodul ce memorează noul atom spre rădăcină, conform următorului algoritm:

1. Presupunem că vârful curent este j. Iniţial avem $j = n - 1$.
2. Dacă $j > 0$, atunci determină $k$ = poziţia nodului tată. Altfel, refacerea proprietăţii este terminată.
3. Dacă a[j] ≤ a[k], atunci refacerea proprietăţii este terminată.
4. Dacă a[j] > a[k], atunci
   (a) Interschimbă a[j] cu a[k].
   (b) Repetă pasul 2 cu vârful curent $j \leftarrow k$.

Descrierea completă a algoritmului de inserare este:

```
procedure insereaza(a, n, o_cheie)
   n ← n+1
   a[n-1] ← o_cheie
   j ← n-1
   esteHeap ← false
   while ((j > 0) and not esteHeap)
      k ← [(j-1)/2]
      if (a[j] > a[k])
         then swap(a[j], a[k])
         else esteHeap ← true
      j ← k
   esteHeap ← true
end
```

Timpul de execuţie în cazul cel mai nefavorabil este $O(\log_2 n)$. În figura 2.16 este arătată inserarea unui atom cu cheia egală cu 20 în max-*heap*-ul din figura 2.14.

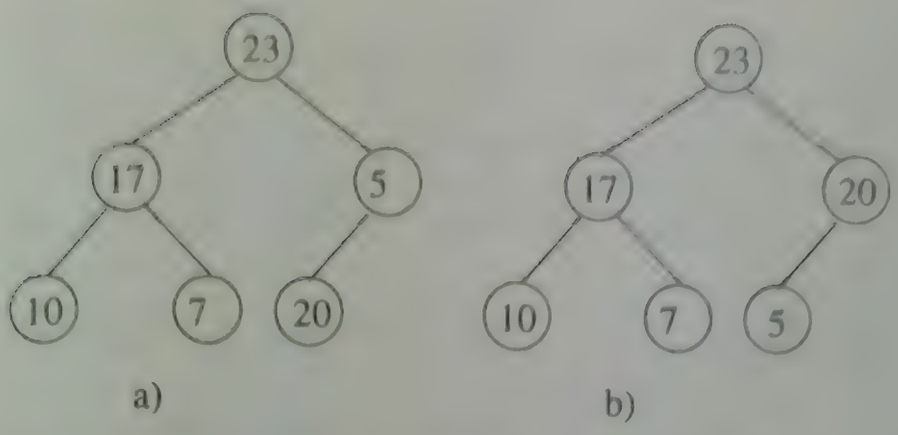



Figura 2.16: Inserarea cheii 20 în max-*heap*-ul din figura 2.14

Citește. Întoarce primul element din tablou. Timpul de execuţie în cazul cel mai nefavorabil este $O(1)$.

### 2.8.3 Exerciţii

**Exerciţiul 2.8.1.** Să se scrie un program care, pentru un tablou a dat, determină dacă a are proprietatea MAX-HEAP.

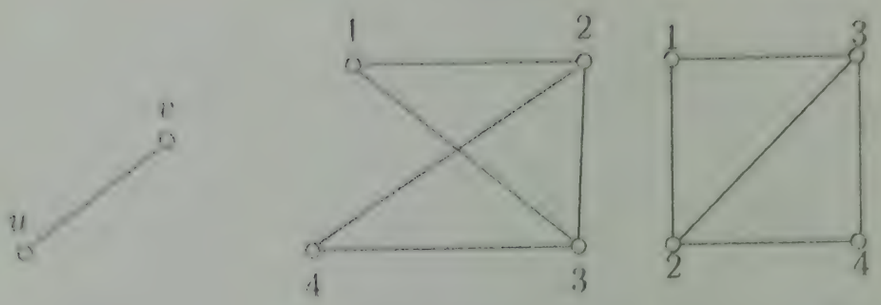
## 2.9 Grafuri

### 2.9.1 Definiţii

Prezentarea de aici se bazează pe cea din [Cro92]. Un *graf* este o pereche $G = (V, E)$, unde $V$ este o mulţime ale cărei elemente le numim *vârfuri*, iar $E$ este o mulţime de perechi neordonate $\{u, v\}$ de vârfuri, pe care le numim *muchii*. Aici considerăm numai cazul când $V$ şi $E$ sunt finite. Dacă $e = \{u, v\}$ este o muchie, atunci $u$ şi $v$ se numesc extremităţile muchiei $e$; mai spunem că $e$ este *incidentă* în $u$ şi $v$ sau că vârfurile $u$ şi $v$ sunt *adiacente (vecine)*. În general, muchiile şi grafurile sunt reprezentate grafic ca în figura 2.17. Dacă $G$ conţine muchii de forma $\{u, u\}$, atunci o asemenea muchie se numeşte *buclă*, iar graful se numeşte *graf general* sau *pseudograf*. Dacă pot să existe mai multe muchii $\{u, v\}$, atunci $G$ se numeşte *multigraf*. Denumirea provine de la faptul că, în acest caz, $E$ este o multimulţime. Considerăm numai cazul în care nu există nici bucle şi nici muchii multiple. Două grafuri $G = (V, E), G' = (V', E')$ sunt *izomorfe* dacă există o funcţie $f : V \to V'$ astfel încât $\{u, v\}$ este muchie în $G$, dacă şi numai dacă $\{f(u), f(v)\}$ este muchie în $G'$. Un graf $G' = (V', E')$ este *subgraf* al lui $G = (V, E)$ dacă $V' \subseteq V, E' \subseteq E$. Un *subgraf parţial* $G'$ al lui $G$ este un subgraf cu proprietatea $V' = V$. Dacă $G$ este un graf şi $X \subseteq V$ o submulţime de vârfuri, atunci *subgraful indus* de $X$ are ca mulţime de vârfuri pe $X$ şi mulţimea de muchii formată din toate muchiile lui $G$ incidente numai în vârfuri din $X$.

Un *mers* de la $u$ la $v$ (de extremităţi $u$ şi $v$) în graful $G$ este o secvenţă

$$u = v_0, \{v_0, v_1\}, v_1, \ldots, \{v_{n-1}, v_n\}, v_n = v,$$




Muchie Două reprezentări ale aceluiași graf

Figura 2.17: Reprezentarea grafurilor

unde $v_i, 0 \leq i \leq n$, sunt vârfuri, iar $\{v_{i-1}, v_i\}, 1 \leq i \leq n$, sunt muchii. Uneori, un mers se precizează numai prin muchiile sale sau numai prin vârfurile sale. În exemplul din figura 2.17, secvența $4, \{4,3\}, 3, \{3,1\}, 1, \{1,2\}, 2, \{2,3\}, 3, \{3,1\}, 1$ este un mers de la vârful 4 la vârful 1. Acest mers mai este notat prin $\{4,3\}, \{3,1\}, \{1,2\}, \{2.3\}, \{3.1\}$ sau prin $4,3,1,2,3,1$. Dacă într-un mers toate muchiile sunt distincte, atunci el se numește *parcurs*, iar dacă toate vârfurile sunt distincte, atunci el se numește *drum*. Mersul din exemplul de mai sus nu este nici parcurs și nici drum. Un mers în care extremitățile coincid se numește *închis* sau *ciclu*. Dacă într-un mers toate vârfurile sunt distincte, cu excepția extremităților, se numește *circuit* (*drum închis*). Un graf $G$ se numește *conex* dacă între oricare două vârfuri ale sale există un drum. Un graf care nu este conex se numește *neconex*. Orice graf se poate exprima ca fiind reuniunea disjunctă de subgrafuri induse, conexe și maximale cu această proprietate; aceste subgrafuri sunt numite *componente conexe*. De fapt, o componentă conexă este subgraful indus de o clasă de echivalență, unde relația de echivalență este dată prin: vârfurile $u$ și $v$ sunt echivalente dacă și numai dacă există un drum de la $u$ la $v$. Graful din figura 2.18 are două componente conexe: $G_1 = (\{1,3,4,6\}, \{\{1,4\}, \{1,6\}, \{4,3\}, \{6,3\}\})$ și $G_2 = (\{2,5,7\}, \{\{2,7\}, \{\{5,2\}, \{\{7,5\}\})$.

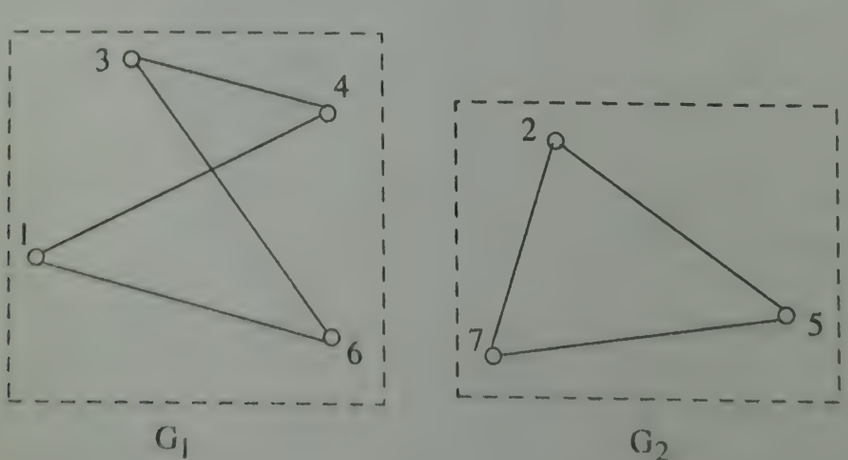
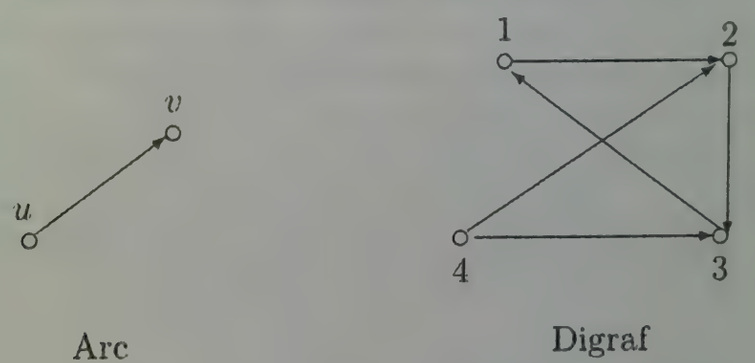



Figura 2.18: Componente conexe

Un *digraf* este o pereche $D = (V, A)$, unde $V$ este o mulțime de vârfuri, iar $A$ este o mulțime de perechi ordonate $(u, v)$ de vârfuri. Considerăm cazul când

$V$ şi $A$ sunt finite. Elementele mulţimii $A$ se numesc *arce*. Dacă $a = (u, v)$ este un arc, atunci spunem că $u$ este *extremitatea iniţială* (*sursa*) a lui $a$ şi că $v$ este *extremitatea finală* (*destinaţia*) lui $a$; mai spunem că $u$ este un *predecesor imediat* al lui $v$ şi că $v$ este un *succesor imediat* al lui $u$. Formulări echivalente: $a$ este *incident din* $u$ şi *incident în (spre)* $v$, sau $a$ este *incident cu* $v$ *spre interior* şi $a$ este *incident cu* $u$ *spre exterior*, sau $a$ *pleacă din* $u$ *şi soseşte în* $v$. Arcele şi digrafurile sunt reprezentate ca în figura 2.19. Orice digraf $D$ defineşte un graf, numit *graf suport al lui* $D$, obţinut prin înlocuirea oricărui arc $(u, v)$ cu muchia $\{u, v\}$ (dacă mai multe arce definesc aceeaşi muchie, atunci se consideră o singură muchie). Graful suport al digrafului din figura 2.19 este cel reprezentat în figura 2.17. Mersurile, parcursurile, drumurile, ciclurile şi circuitele se definesc ca în cazul grafurilor, dar considerând arce în loc de muchii. Un digraf se numeşte *tare conex* dacă pentru orice două vârfuri $u$ şi $v$, există un drum de la $u$ la $v$ şi un drum de la $v$ la $u$. O *componentă tare conexă* este subgraful indus de o clasă de echivalenţă, unde relaţia de echivalenţă invocă acum definiţia de drum într-un digraf. Digraful din figura 2.20 are trei componente tare conexe: $G_1 = (\{1, 3\}, \{(1, 3), (3, 1)\})$, $G_2 = (\{2\}, \emptyset)$, $G_3 = (\{4, 5, 6\}, \{(4, 5), (5, 6), (6, 4)\})$. Un digraf $D$ este *conex* dacă graful suport al lui $D$ este conex.




Arc Digraf

Figura 2.19: Reprezentarea digrafurilor

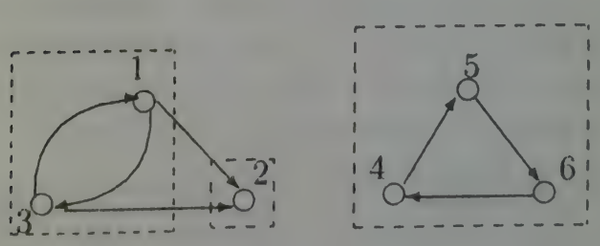



Figura 2.20: Componente tare conexe

Un *(di)graf etichetat pe vârfuri* este un (di)graf $G = (V, E)$ împreună cu o mulţime de etichete $L$ şi o funcţie de etichetare $\ell : V \to L$. În figura 2.21.a este reprezentat un graf etichetat, în care funcţia de etichetare este $\ell(1) = A, \ell(2) = B, \ell(3) = C$. Un *(di)graf etichetat pe muchii (arce)* este un (di)graf $G = (V, E)$ împreună cu o mulţime de etichete $L$ şi o funcţie de etichetare $\ell : E \to L$. În figura 2.21.b este reprezentat un graf etichetat pe arce, în care funcţia de etichetare este $\ell(\{1, 2\}) = A, \ell(\{2, 3\}) = B, \ell(\{3, 1\}) = C$. Dacă mulţimea de etichete este o

submulţime a lui $\mathcal{R}$ (mulţimea numerelor reale), atunci (di)graful se mai numeşte *ponderat*.

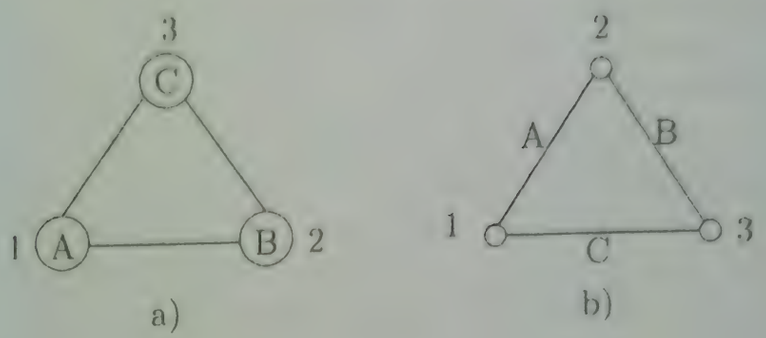



Figura 2.21: Grafuri etichetate

Un *arbore* este un graf conex fără circuite. Un *arbore cu rădăcină* este un digraf fără circuite, în care există un vârf $r$, numit *rădăcină*, cu proprietatea că, pentru orice alt vârf $v$, există un singur drum de la $r$ la $v$. Într-un arbore cu rădăcină, sensurile arcelor sunt unic determinate de drumurile care pleacă din rădăcină şi din acest motiv nu vom mai desena săgeţile care marchează orientarea. Un *arbore cu rădăcină ordonat* este un arbore cu rădăcină cu proprietatea că orice vârf $u$, exceptând rădăcina, are exact un predecesor imediat şi, pentru orice vârf, mulţimea succesorilor imediaţi este total ordonată. Într-un arbore cu rădăcină ordonat, succesorii imediaţi ai vârfului $u$ se numesc şi *fii* ai lui $u$, iar predecesorul imediat al lui $u$ se numeşte *tatăl* lui $u$. Un caz particular de arbore ordonat este arborele binar, unde relaţia de ordine peste mulţimea fiilor vârfului $u$ este precizată prin (fiul stâng, fiul drept). Exemple de arbori sunt date în figura 2.22.

## 2.9.2 Tipul de dată abstract Graf

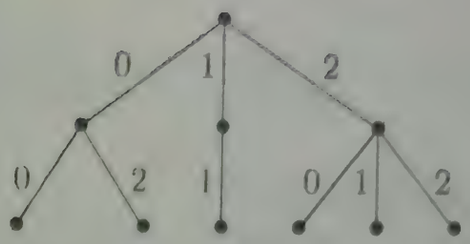
### 2.9.2.1 Descrierea obiectelor

Obiectele de tip dată sunt grafuri $G = (V, E)$, definite în mod unic până la un izomorfism, unde mulţimea de vârfuri $V$ este o submulţime finită a tipului abstract Vârf, iar mulţimea de muchii $E$ este o submulţime a tipului abstract Muchie. Fără să restrângem generalitatea, presupunem că mulţimile de vârfuri $V$ sunt mulţimi de forma $\{0, 1, \ldots, n-1\}$, unde $n = \#V$, iar mulţimile de muchii sunt submulţimi ale mulţimii $\{\{i, j\} \mid i, j \in \{0, 1, \ldots, n-1\}\}$. Dacă $G = (V, E)$ este oarecare cu $\#V = n$, atunci putem defini o funcţie bijectivă $index_G : V \to \{0, 1, \ldots, n-1\}$ şi graful $G' = (\{0, 1, \ldots, n-1\}, E')$ cu $\{index_G(u), index_G(v)\} \in E' \iff \{u, v\} \in E$. Evident, $G$ şi $G'$ sunt izomorfe şi deci putem lucra cu $G'$ în loc de $G$. Întoarcerea de la $G'$ la $G$ se poate face via funcţia inversă lui $index_G$, $nume_G : \{0, 1, \ldots, n-1\} \to V$, dată prin $nume_G(i) = v \iff index_G(v) = i$.

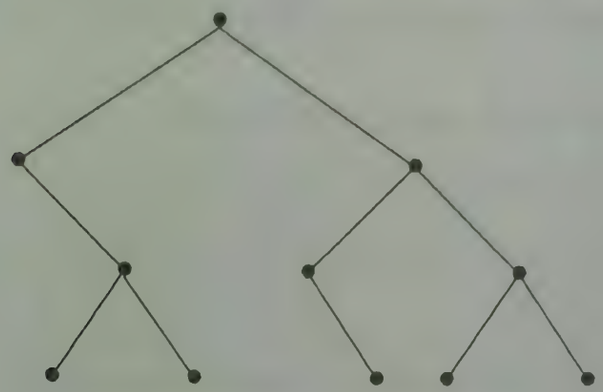
### 2.9.2.2 Operaţii

GrafVid.
*Intrare:* – nimic;
*Ieşire:* – graful vid $G = (\emptyset, \emptyset)$.

Arbore




Arbore cu rădăcină ordonat



Arbore binar

Figura 2.22: Exemple de arbori

EsteGrafVid.

*Intrare:* – un graf $G = (V, E)$;
*Ieşire:* – *true* dacă $G$ este vid,
– *false* în caz contrar.

Inserează Vârf.

*Intrare:* – un graf $G = (V, E)$ cu $V = \{0, 1, \ldots, n-1\}$;
*Ieşire:* graful $G$ la care se adaugă vârful $n$ ca vârf izolat (nu există muchii incidente în $n$).

Inserează Muchie.

*Intrare:* – un graf $G = (V, E)$ şi două vârfuri diferite $i, j \in V$;
*Ieşire:* – graful $G$ la care se adaugă muchia $\{i, j\}$.

Şterge Vârf.

*Intrare:* – un graf $G = (V, E)$ şi un vârf $k \in V$;
*Ieşire:* graful $G$ din care au fost eliminate toate muchiile incidente în $k$ (au o extremitate în $k$), iar vârfurile $i > k$ sunt redenumite $i - 1$.

Şterge Muchie.

*Intrare:* – un graf $G = (V, E)$ şi două vârfuri diferite $i, j \in V$;
*Ieşire:* – graful $G$ din care a fost eliminată muchia $\{i, j\}$.

ListaDeAdiacență.

*Intrare:* – un graf $G = (V, E)$ și un vârf $i \in V$;
*Ieșire:* – mulțimea vârfurilor adiacente cu vârful $i$.

ListaVârfurilorAccesibile.

*Intrare:* – un graf $G = (V, E)$ și un vârf $i \in V$;
*Ieșire:* – mulțimea vârfurilor accesibile din vârful $i$. Un vârf $j$ este accesibil din $i$ dacă și numai dacă există un drum de la $i$ la $j$.

## 2.9.3 Tipul de dată abstract Digraf

### 2.9.3.1 Descrierea obiectelor

Obiectele de tip dată sunt digrafuri $D = (V, A)$, unde mulțimea de vârfuri $V$ o considerăm de forma $\{0, 1, \ldots, n-1\}$, iar mulțimea de arce este o submulțime a produsului cartezian $V \times V$. Tipul Vârf are aceeași semnificație ca în cazul modelului constructiv Graf, iar tipul Arc este produsul cartezian Vârf × Vârf.

### 2.9.3.2 Operații

DigrafVid.

*Intrare:* – nimic;
*Ieșire:* – digraful vid $D = (\emptyset, \emptyset)$.

EsteDigrafVid.

*Intrare:* – un digraf $D = (V, A)$;
*Ieșire:* – *true*, dacă $D$ este vid,
– *false*, în caz contrar.

Inserează Vârf.

*Intrare:* – un digraf $D = (V, A)$ cu $V = \{0, 1, \ldots, n-1\}$;
*Ieșire:* – digraful $D$ la care s-a adăugat vârful $n$.

Inserează Arc.

*Intrare:* – un digraf $D = (V, A)$ și două vârfuri diferite $i, j \in V$;
*Ieșire:* – digraful $D$ la care s-a adăugat arcul $(i, j)$.

ȘtergeVârf.

*Intrare:* – un digraf $D = (V, A)$ și un vârf $k \in V$;
*Ieșire:* – digraful $D$ din care au fost eliminate toate arcele incidente în $k$ (au o extremitate în $k$), iar vârfurile $i > k$ sunt redenumite $i - 1$.

ȘtergeArc.

*Intrare:* – un digraf $D = (V, A)$ și două vârfuri diferite $i, j \in V$;
*Ieșire:* – digraful $D$ din care s-a eliminat arcul $(i, j)$.

ListaDeAdiacenţăInterioară.

*Intrare:* un digraf $D = (V, A)$ şi un vârf $i \in V$;
*Ieşire:* mulţimea surselor (vârfurilor de plecare ale) arcelor care sosesc în vârful $i$.

ListaDeAdiacenţăExterioară.

*Intrare:* un digraf $D = (V, A)$ şi un vârf $i \in V$;
*Ieşire:* mulţimea destinaţiilor (vârfurilor de sosire ale) arcelor care pleacă din vârful $i$.

ListaVârfurilorAccesibile.

*Intrare:* un graf $D = (V, A)$ şi un vârf $i \in V$;
*Ieşire:* mulţimea vârfurilor accesibile din vârful $i$. Un vârf $j$ este accesibil din $i$ dacă şi numai dacă există un drum de la $i$ la $j$.

### 2.9.4 Reprezentarea grafurilor ca digrafuri

Orice graf $G = (V, E) \in$ Graf poate fi reprezentat ca un digraf $D = (V, A) \in$ Digraf considerând pentru fiecare muchie $\{i, j\} \in E$ două arce $(i, j), (j, i) \in A$. Cu aceste reprezentări, operaţiile tipului Graf pot fi exprimate cu ajutorul celor ale tipului Digraf. Astfel, inserarea/ştergerea unei muchii este echivalentă cu inserarea/ştergerea a două arce. Această reprezentare ne va permite să ne ocupăm în continuare numai de implementări ale tipului abstract Digraf.

### 2.9.5 Implementarea cu matrice de adiacenţă

#### 2.9.5.1 Reprezentarea obiectelor

Digraful $D$ este reprezentat printr-o structură cu trei câmpuri: `D.n` = numărul de vârfuri, `D.m` = numărul de arce şi un tablou bidimensional `D.a` de dimensiune $n \times n$ astfel încât:

$$D.a[i, j] = \begin{cases} 0 & \text{dacă } (i, j) \notin A, \\ 1 & \text{dacă } (i, j) \in A. \end{cases}$$

pentru $i, j = 0, \ldots, n-1$. Dacă `D` este reprezentarea unui graf, atunci matricea de adiacenţă este simetrică:

$$D.a[i, j] = D.a[j, i], \text{ pentru orice } i, j.$$

**Observaţie.** În loc de valorile 0 şi 1 se pot considera valorile booleene *false* şi respectiv *true*. sfobs

#### 2.9.5.2 Implementarea operaţiilor

DigrafVid. Este reprezentat de orice variabilă `D` cu $D.n = 0$ şi $D.m = 0$.

EsteDigrafVid. Se testează dacă $D.m$ şi $D.n$ sunt egale cu zero.

Inserează Vârf. Constă în adăugarea unei linii şi a unei coloane la matricea de adiacenţă.

```
procedure insereazaVarf(D)
   D.n ← D.n+1
   for i ← 0 to D.n-1 do
       D.a[i,D.n-1] ← 0
       D.a[D.n-1,i] ← 0
end
```

Inserează Arc Inserarea arcului $(i, j)$ presupune numai actualizarea componentei D.a[i, j], adică D.a[i, j] ← 1.

Şterge Vârf. Notăm cu $a$ valoarea matricei D.a înainte de ştergerea vârfului $k$ şi cu $a'$ valoarea de după ştergere. Au loc următoarele relaţii (vezi şi figura 2.23):

1. dacă $i, j < k$ atunci $a'_{i,j} = a_{i,j}$;
2. dacă $i < k \leq j$ atunci $a'_{i,j} = a_{i,j+1}$;
3. dacă $j < k \leq i$ atunci $a'_{i,j} = a_{i+1,j}$;
4. dacă $k \leq i, j$ atunci $a'_{i,j} = a_{i+1,j+1}$.

Din aceste relaţii deducem următorul algoritm:

```
procedure eliminaVirf(D, k)
   for i ← k+1 to D.n-1 do
      for j ← 0 to D.n-1 do
         D.a[i-1, j] ← D.a[i,j]
   for j ← k+1 to D.n-1 do
      for i ← 0 to D.n-1 do
         D.a[i, j-1] ← D.a[i,j]
   D.n ← D.n-1
end
```

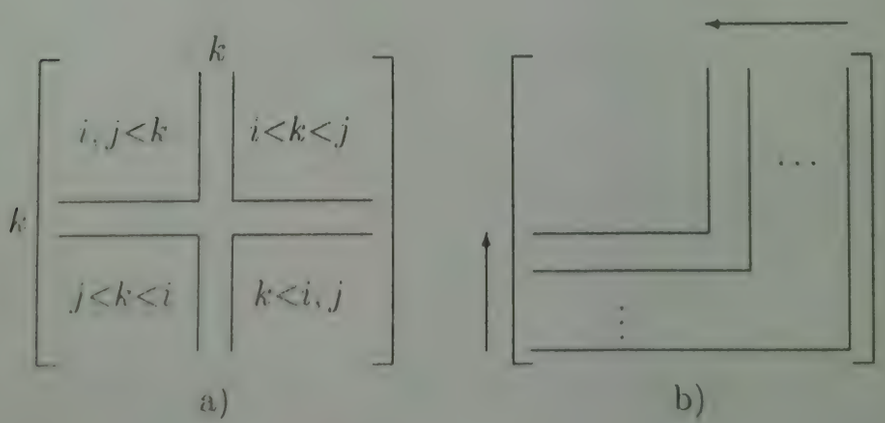



Figura 2.23: Ştergerea unui vârf

Şterge Arc. Asemănător operaţiei Inserează Arc.

ListaDeAdiacenţăInterioară şi ListaDeAdiacentăExterioară. Lista vârfurilor destinatare ale arcelor care „pleacă" din $i$ este reprezentată de linia $i$, iar lista vârfurilor sursă ale arcelor care sosesc în $i$ este reprezentată de coloana $i$. Dacă `D` este reprezentarea unui graf, atunci lista vârfurilor adiacente se obţine numai prin consultarea liniei (sau numai a coloanei) $i$.

ListaVârfurilorAccesibile. Reamintim că un vârf $j$ este accesibil în $D$ din $i$ dacă există în $D$ un drum de la $i$ la $j$. Dacă $i = j$, atunci, evident, $j$ este accesibil din $i$ (există un drum de lungime zero). Dacă $i \neq j$, atunci există drum de la $i$ la $j$ dacă există arc de la $i$ la $j$ sau există $k$ cu proprietatea că există drum de la $i$ la $k$ şi drum de la $k$ la $j$. De fapt, cele de mai sus sunt echivalente cu afirmaţia că relaţia „există drum de la $i$ la $j$", definită peste $V$, este închiderea reflexivă şi tranzitivă a relaţiei „există arc de la $i$ la $j$". Ultima relaţie este reprezentată de matricea de adiacenţă, iar prima relaţie se poate obţine din ultima prin următorul algoritm datorat lui Warshall (1962):

```
procedure detInchReflTranz(D, b)
   for i ← 0 to D.n-1 do
       for j ← 0 to D.n-1 do
           b[i,j] ← D.a[i,j]
           if (i = j) then b[i,j] ← 1
   for k ← 0 to D.n-1 do
       for i ← 0 to D.n-1 do
           if (b[i,k] = 1)
              then for j ← 0 to D.n-1 do
                       if (b[k,j] = 1) then b[i,j] ← 1
end
```

Lista vârfurilor accesibile din vârful $i$ este reprezentată acum de linia $i$ a tabloului bidimensional `b`. În continuare vom arăta că subprogramul `detInchReflTranz` determină într-adevăr închiderea reflexivă şi tranzitivă. Se observă uşor că reflexivitatea este rezolvată de primele două instrucţiuni `for` care realizează şi copierea `D.a` în `b`. În continuare ne ocupăm de tranzitivitate. Fie $i$ şi $j$ două vârfuri cu proprietatea că $j$ este accesibil din $i$. Există un $k$ şi un drum de la $i$ la $j$ cu vârfuri intermediare din mulţimea $X = \{0, \ldots, k\}$. Vom arăta că după $k$ iteraţii avem $b[i, j] = 1$. Procedăm prin inducţie după $\#X$. Dacă $X = \emptyset$, atunci $b[i, j] = D.a[i, j] = 1$. Presupunem $\#X > 0$. Rezultă că există un drum de la $i$ la $k$ cu vârfuri intermediare din $\{0, \ldots, k-1\}$ şi un drum de la $k$ la $j$ cu vârfuri intermediare tot din $\{0, \ldots, k-1\}$. Din ipoteza inductivă, rezultă că după $k-1$ execuţii ale buclei `for k ...` avem $b[i, k] = 1$ şi $b[k, j] = 1$. După cea de-a $k$-a execuţie al buclei obţinem $b[i, j] = 1$, prin execuţia ramurii `then` a ultimei instrucţiuni `if`.

## 2.9.6 Implementarea cu liste de adiacenţă înlanţuite

### 2.9.6.1 Reprezentarea obiectelor

Un digraf $D$ este reprezentat printr-o structură asemănătoare cu cea de la matricele de adiacenţă, dar matricea de adiacenţă este înlocuită cu un tablou unidimensional

de $n$ liste liniare, implementate prin liste simplu înlănţuite şi notate cu D.a[i] pentru $i = 0, \ldots, n-1$, astfel încât lista D.a[i] conţine vârfurile destinatare ale arcelor care pleacă din $i$ (= lista de adiacenţă exterioară).

**Exemplu.** Fie $G$ graful reprezentat în figura 2.24a. Tabloul listelor de adiacenţă corepunzătoare lui $G$ este reprezentat în figura 2.24b. Pentru digraful $D$ din figura 2.25.a, tabloul listelor de adiacenţă înlănţuite este reprezentat în figura 2.25.b. sfex

De multe ori este util ca, atunci când peste mulţimea vârfurilor este dată o relaţie de ordine, componentele listelor de adiacenţă să respecte această ordine.

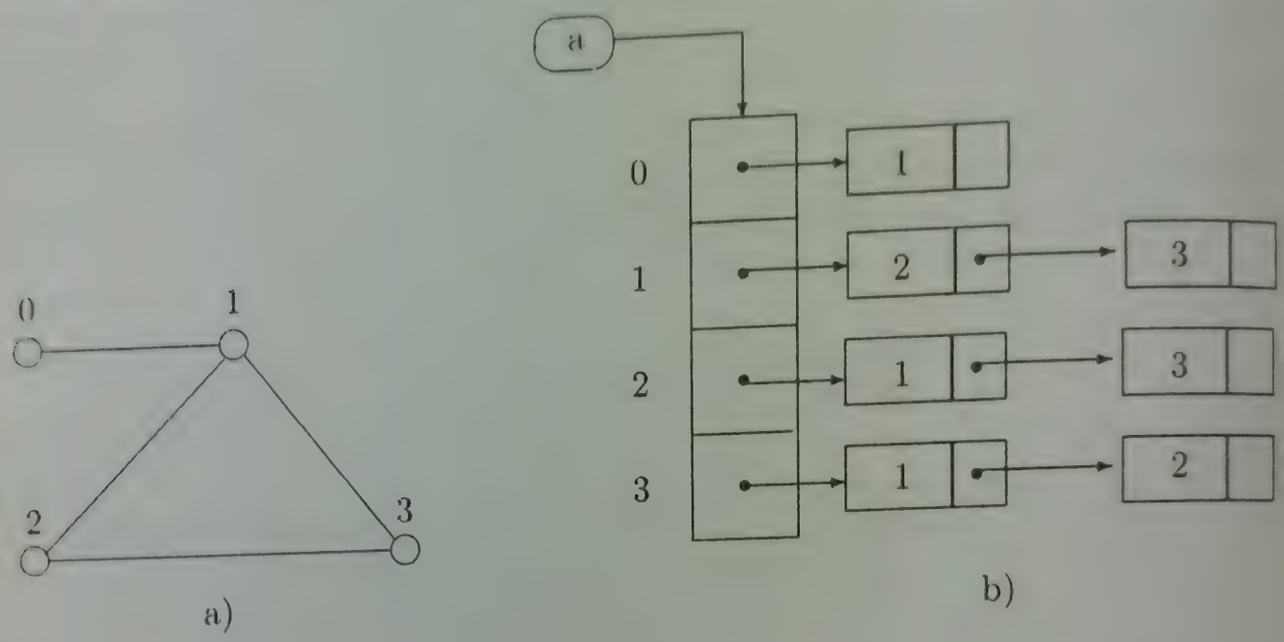



Figura 2.24: Graf reprezentat prin liste de adiacenţă înlănţuite

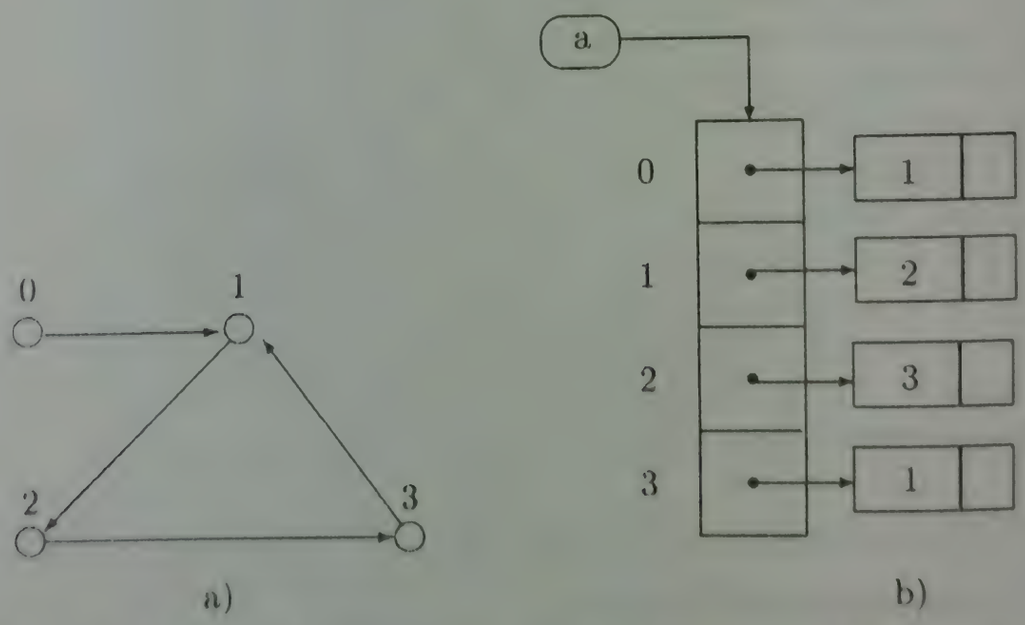



Figura 2.25: Digraf reprezentat prin liste de adiacenţă înlănţuite

#### 2.9.6.2 Implementarea operaţiilor

DigrafVid. Similar celei de la implementarea cu matrice de adiacenţă.

EsteDigrafVid. Similar celei de la implementarea cu matrice de adiacenţă.

Inserează Vârf. Se incrementează `D.n` şi se iniţializează `D.a[n]` cu lista vidă:

```
D.n ← D.n+1
D.a[D.n-1] ← listaVida()
```

Inserează Arc. Adăugarea arcului $(i, j)$ constă în inserarea unui nou nod în lista `D.a[i]` în care se memorează valoarea $j$:

```
insereaza(D.a[i], 0, j)
```

Timpul de execuţie în cazul cel mai nefavorabil este $O(1)$.

Şterge Vârf. Ştergerea vârfului $i$ constă în eliminarea listei `D.a[i]` din tabloul `D.a` şi parcurgerea tuturor listelor pentru a elimina nodurile care memorează $i$ şi pentru a redenumi vârfurile $j > i$ cu $j - 1$.

```
procedure stergeVarf(D, k)
   while (D.a[k].prim ≠ NULL) do
      p ← D.a[k].prim
      D.a[k].prim ← p->succ
      delete(p)
      D.a[k].ultim ← NULL
   D.n ← D.n-1
   for i ← k to D.n-1 do
       D.a[i] ← D.a[i+1]
   for i ← 0 to D.n-1 do
       elimina*(D.a[i], k) /* elimină k şi redenumeşte vârfurile */
end
```

Timpul de execuţie în cazul cel mai nefavorabil este $O(m)$, unde $m$ este numărul de arce din digraf ($m \leq n^2$).

Şterge Arc. Ştergerea arcului $(i, j)$ constă în eliminarea nodului care memorează valoarea $j$ din lista `D.a[i]`.

```
procedure stergeArc(D, i, j)
   elimina(D.a[i], j)
end
```

Timpul de execuţie în cazul cel mai nefavorabil este $O(n)$.

ListaDeAdiacenţăInterioară. Determinarea listei de adiacenţă interioară pentru vârful $i$ constă în selectarea acelor vârfuri $j$ cu proprietatea că $i$ apare în lista `D.a[j]`. Presupunem că lista de adiacenţă interioară este reprezentată de o coadă `C`:

```
procedure listaAdInt(D, i, C)
   C ← coadaVida()
   for j ← 0 to D.n-1 do
      p ← D.a[j].prim
      while (p ≠ NULL) do
         if (p->elt = i)
            then insereaza(C, j)
                 p ← NULL
         p ← p->succ
end
```

Timpul de execuţie în cazul cel mai nefavorabil este $O(m)$, unde $m$ este numărul de arce din digraf ($m \leq n^2$).

ListaDeAdiacenţăExterioară. Lista de adiacenţă exterioară a vârfului $i$ este `D.a[i]`.

ListaVârfurilorAccesibile. Notăm cu $i_0$ vârful din (di)graful $D$ pentru care se doreşte să se calculeze lista vârfurilor accesibile. Algoritmul pe care-l propunem va impune un proces de vizitare succesivă a vecinilor imediaţi lui $i_0$, apoi a vecinilor imediaţi ai acestora şi aşa mai departe. În timpul procesului de vizitare vor fi gestionate două mulţimi:

- $S$ - mulţimea vârfurilor accesibile din $i_0$ vizitate până în acel moment;
- $SB$ – o submulţime de vârfuri din $S$ pentru care este posibil să existe vârfuri adiacente accesibile din $i_0$ nevizitate încă.

Vom utiliza, de asemenea, un tablou de pointeri ($\mathtt{p[i]} \mid 0 \leq i < n$) cu care ţinem evidenţa primului vârf nevizitat încă din fiecare listă de adiacenţă. Mai precis, dacă $p[i] \neq NULL$ şi $i \in S$, atunci $i \in SB$. Algoritmul constă în execuţia repetată a următorului proces:

- se alege un vârf $i \in SB$;
  dacă primul element neprocesat încă din lista `D.a[i]` nu este în $S$, atunci se adaugă atât la $S$, cât şi la $SB$;
- dacă lista `D.a[i]` a fost parcursă complet, atunci elimină $i$ din $SB$.

Descrierea schematică a algoritmului de explorare a unui digraf este:

```
procedure explorareDigraf(D, i0, viziteaza(), S)
   for  ← 0 to D.n-1 do
      p[i]  ← D.a[i].prim
   S ← {i0}
   SB ← {i0}
   viziteaza(i0)
   while (SB ≠ ∅) do
      i ← citeste(SB)
      if (p[i] = NULL)
```

```
            then SB ← SB \ {i}
            else j ← p[i]->elt
                 p[i] ← p[i]->succ
                 if j ∉ S
                    then viziteaza(j)
                         S ← S ∪ {j}
                         SB ← SB ∪ {j}
end
```

**Teorema 2.2.** *Procedura* `explorareDigraf` *determină vârfurile accesibile din* $i_0$ *în timpul* $O(\max(n, m_0))$, *unde* $m_0$ *este numărul muchiilor accesibile din* $i_0$, *şi utilizează spaţiul* $O(\max(n, m))$.

*Demonstraţie.* Corectitudinea rezultă din faptul că următoarea proprietate este invariantă:

> $S$ conţine o submulţime de noduri accesibile din $i_0$ vizitate deja, $SB \subseteq S$ şi mulţimea vârfurilor accesibile din $i_0$ nevizitate încă este inclusă în mulţimea vârfurilor accesibile din $SB$.

Timpul de execuţie $O(\max(n, m_0))$ rezultă în ipoteza că operaţiile asupra mulţimilor $S$ şi $SB$ se realizează în timpul $O(1)$. Se observă imediat că partea de iniţializare necesită $O(n)$ timp, iar bucla `while` se repetă de $O(m_0)$ ori. Dimensiunea spaţiului de memorie rezultă imediat din declaraţii. sfdem

În funcţie de structura de date utilizată pentru gestionarea mulţimii $SB$, obţinem diferite strategii de explorare a digrafurilor.

**Explorarea DFS (Depth First Search).** Se obţine din `explorareDigraf` prin reprezentarea mulţimii `SB` printr-o stivă. Presupunem că mulţimea $S$ este reprezentată prin vectorul său caracteristic:

$$S[i] = \begin{cases} 1, & \text{dacă } i \in S \\ 0, & \text{altfel.} \end{cases}$$

Înlocuim în procedura `explorareDigraf` operaţiile corespunzătoare lui `SB` cu operaţii caracteristice stivei, iar operaţiile corespunzătoare lui `S` cu operaţii caracteristice tablourilor. Obţinem astfel procedura `DFS` care descrie strategia DFS:

```
procedure DFS(D, i0, viziteaza(), S)
   for i ← 0 to D.n-1 do
       p[i] ← D.a[i].prim
       S[i] ← 0
   SB ← stivaVida()
   push(SB, i0)
   S[i0] ← 1
   viziteaza(i0)
   while (not esteStivaVida(SB)) do
      i ← top(SB)
```

```
        if (p[i] = NULL)
           then pop(SB)
           else j ← p[i]->elt
                p[i] ← p[i]->succ
                if S[j] = 0
                   then viziteaza(j)
                        S[j] ← 1
                        push(SB, j)
    end
```

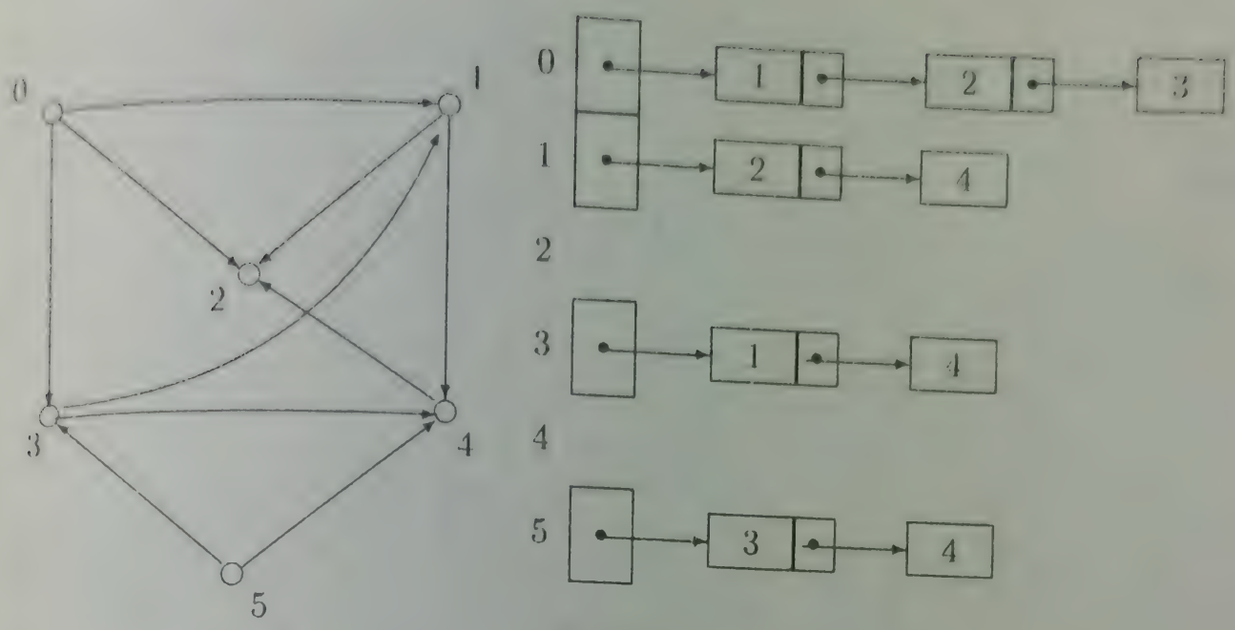
Notăm faptul că implementarea respectă cerinţa ca operaţiile peste mulţimile $S$ şi $SB$ să se execute în timpul O(1).

**Exemplu.** Considerăm digraful din figura 2.27. Presupunem $i_0 = 0$. Calculul procedurii *DFS* este descris în tabelul din figura 2.26. Descrierea pe scurt a acestui calcul este: se vizitează vârful 0, apoi primul din lista vârfului 0 – adică 1, după care primul din lista lui 1 – adică 2. Pentru că 2 nu are vârfuri adiacente spre exterior, se revine la 1 şi vizitează al doilea din lista vârfului 1 - adică 4. Analog vârfului 2, pentru că 4 nu are vârfuri adiacente spre exterior se revine la 1 şi pentru că lista lui 1 este epuizată se revine la lista lui 0. Al doilea din lista lui 0 este 2, dar acesta a fost deja vizitat şi deci nu mai este luat în considerare. Următorul din lista lui 0 este vârful 3 care nu a mai fost vizitat, după care se ia în considerare primul din lista lui 3 – adică 1, însă acesta a mai fost vizitat. La fel şi următorul din lista lui 3 – adică 4. Aşadar, lista ordonată dată de explorarea DFS a vârfurilor accesibile din 0 este: $(0, 1, 2, 4, 3)$. sfex

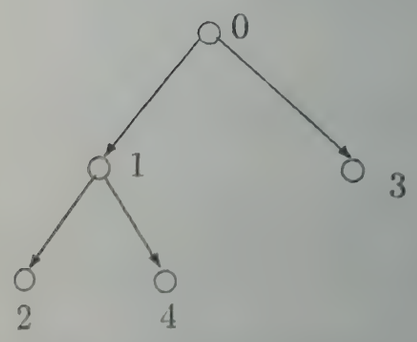
| $i$ | $j$ | $S$ | $SB$ |
|---|---|---|---|
| | | $\{0\}$ | $(0)$ |
| 0 | 1 | $\{0,1\}$ | $(0,1)$ |
| 1 | 2 | $\{0,1,2\}$ | $(0,1,2)$ |
| 2 | -- | $\{0,1,2\}$ | $(0,1)$ |
| 1 | 4 | $\{0,1,2,4\}$ | $(0,1,4)$ |
| 4 | -- | $\{0,1,2,4\}$ | $(0,1)$ |
| 1 | -- | $\{0,1,2,4\}$ | $(0)$ |
| 0 | 2 | $\{0,1,2,4\}$ | $(0)$ |
| 0 | 3 | $\{0,1,2,3,4\}$ | $(0,3)$ |
| 3 | 1 | $\{0,1,2,3,4\}$ | $(0,3)$ |
| 3 | 4 | $\{0,1,2,3,4\}$ | $(0,3)$ |
| 3 | | $\{0,1,2,3,4\}$ | $(0)$ |
| 0 | | $\{0,1,2,3,4\}$ | $()$ |

Figura 2.26: Un calcul al procedurii DFS

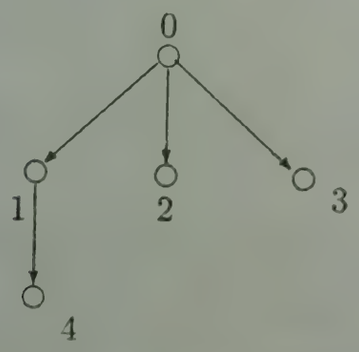
Metodei îi putem asocia un arbore, numit *arbore parţial DFS*, în modul următor: Notăm cu $T$ mulţimea arcelor $(i, j)$ cu proprietatea că $j$ este vizitat prima dată parcurgând acest arc. Pentru a obţine $T$ este suficient să înlocuim în procedura DFS apelul `Viziteaza(j)` cu o instrucţiune de forma `adauga (i,j) la T`. Arborele




a) Digraf




b) Arbore parţial DFS




c) Arbore parţial BFS

Figura 2.27: Explorarea unui digraf

parţial DFS este $S(D, i_0) = (V_0, T)$, unde $V_0$ este mulţimea vârfurilor accesibile din $i_0$. Definiţia este valabilă şi pentru cazul când argumentul procedurii `DFS` este reprezentarea unui graf, doar cu precizarea că se consideră muchii în loc de arce. În figura 2.27.a este reprezentat arborele parţial DFS pentru digraful din figura 2.27.b şi vârful 0. Arborele parţial DFS este util în multe aplicaţii ce necesită parcurgeri de grafuri.

**Explorarea BFS (Breadth First Search).** Se obţine din `ExplorareGraf` prin reprezentarea mulţimii $SB$ printr-o coadă.

**Exerciţiul 2.9.1.** Să se înlocuiască în procedura `ExplorareGraf` operaţiile corespunzătoare lui `SB` cu operaţii caracteristice cozii, iar operaţiile corespunzătoare lui `S` cu operaţii caracteristice tablourilor. Noua procedură se va numi `BFS`.

**Exemplu.** Considerăm digraful din exemplul precedent. Presupunem de asemenea $i_0 = 1$. Calculul procedurii $BFS$ este descris în tabelul din figura 2.28. Descrierea sumară a acestui calcul este: se vizitează vârful 0, apoi toate vârfurile din lista lui 0, apoi toate cele nevizitate din lista lui 1, apoi cele nevizitate din lista lui

| $i$ | $j$ | $S$ | $SB$ |
|---|---|---|---|
| | | $\{0\}$ | $(0)$ |
| 0 | 1 | $\{0,1\}$ | $(0,1)$ |
| 0 | 2 | $\{0,1,2\}$ | $(0,1,2)$ |
| 0 | 3 | $\{0,1,2,3\}$ | $(0,1,2,3)$ |
| 0 | – | $\{0,1,2,3\}$ | $(1,2,3)$ |
| 1 | 2 | $\{0,1,2,3\}$ | $(1,2,3)$ |
| 1 | 4 | $\{0,1,2,3,4\}$ | $(1,2,3,4)$ |
| 1 | – | $\{0,1,2,3,4\}$ | $(2,3,4)$ |
| 2 | – | $\{0,1,2,3,4\}$ | $(3,4)$ |
| 3 | 1 | $\{0,1,2,3,4\}$ | $(3,4)$ |
| 3 | 4 | $\{0,1,2,3,4\}$ | $(3,4)$ |
| 3 | – | $\{0,1,2,3,4\}$ | $(4)$ |
| 4 | – | $\{0,1,2,3,4\}$ | $(\,)$ |

Figura 2.28: Un calcul al procedurii BFS

2 şi aşa mai departe. Lista ordonată dată de parcurgerea BFS este $(0,1,2,3,4)$.

sfex

Ca şi în cazul parcurgerii DFS, metodei i se poate ataşa un arbore, numit *arbore parţial BFS*. În figura 2.27.c este reprezentat arborele parţial BFS din exemplul de mai sus.

Sugerăm cititorului să testeze cele două strategii pe mai multe exemple pentru a vedea clar care este diferenţa dintre ordinile de parcurgere a vârfurilor.

**Exerciţiul 2.9.2.** Am văzut că, în cazul reprezentării digrafurilor prin matricea de adiacenţă, listele de adiacenţă exterioară sunt incluse în liniile matricei. Să se rescrie subprogramele `DFS` şi `BFS` pentru cazul când digraful este reprezentat prin matricea de adiacenţă.

### 2.9.7 Implementarea cu liste de adiacenţă reprezentate cu tablouri

Listele de adiacenţă ale digrafului $D = \langle V, A \rangle$ sunt reprezentate prin două tablouri `D.a[i]`, $i = 0, \ldots, D.n$, şi `D.s[j]`, $j = 0, \ldots, D.m$, care satisfac proprietăţile:

- `D.a[i]` este adresa $j$ în tabloul `D.s` a primului element cu proprietatea $(i, s[j]) \in A$;
- vârfurile destinatare ale arcelor care pleacă din $i$ sunt memorate în componente consecutive din `D.s`;
- `D.a[0]` $=$ `0` şi `D.a[D.n]` $=$ `D.m`.

**Exemplu.** În figura 2.29 este dat un exemplu de digraf reprezentat prin liste de acest fel. Din vârful 0 pleacă trei arce cu destinaţiile 1,2 şi 3 – ce sunt memorate în primele trei componente ale tabloului `D.s`. Din vârful 1 pleacă două arce cu destinaţiile 2 şi 3 – ce sunt memorate în `D.s[3]` şi `D.s[4]`. Din vârful 2 nu pleacă nici

un arc şi de aceea D.a[2] = D.a[3]. Din vârful 3 pleacă un singur arc, memorat în D.a[5]. Elementul D.a[4] are rolul de a marca locul unde se termină lista de adiacenţă exterioară a vârfului 3.



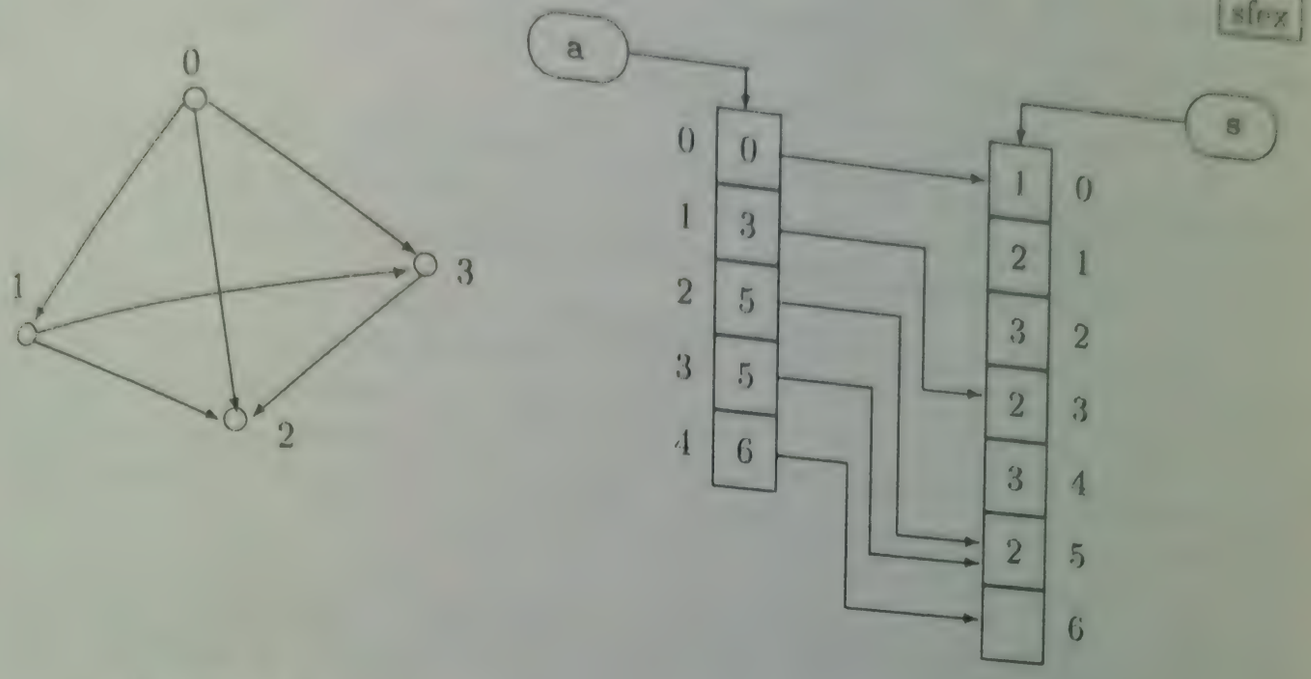



Figura 2.29: Digraf reprezentat prin liste de adiacenţă tablouri

**Exerciţiul 2.9.3.** Să se descrie implementările operaţiilor tipului Digraf pentru cazul când digrafurile sunt implementate cu liste de adiacenţă reprezentate prin tablouri.

## 2.9.8 Exerciţii

**Exerciţiul 2.9.4.** Dacă $G$ este un graf atunci gradul $\delta(i)$ al unui vârf $i$ este egal cu numărul de vârfuri adiacente cu $i$. Dacă $D$ este un digraf, atunci gradul extern ($\delta^+(i)$) al unui vârf $i$ este egal cu numărul de vârfuri adiacente cu $i$ spre exterior, iar gradul intern ($\delta^-(i)$) este numărul de vârfuri adiacente cu $i$ spre interior.

Să se scrie o procedură `detGrad(D, tip, din, dout)` care determină gradele vârfurilor digrafului $D$ (cazul când $tip = true$) sau ale grafului reprezentat de D (cazul când $tip = false$).

**Exerciţiul 2.9.5.** O alegere de drumuri care unesc toate perechile de vârfuri conectabile $i, j$ într-un digraf poate fi memorată într-o matrice $p$ cu semnificaţia: $p[i, j]$ este primul vârf intermediar întâlnit după $i$ pe drumul de la $i$ la $j$.

1. Să se modifice subprogramul `detInchReflTranz` astfel încât să determine şi o alegere de drumuri care unesc vârfurile conectabile.
2. Să se scrie un subprogram care, având date matricea drumurilor şi două vârfuri $i, j$, determină un drumul de la $i$ la $j$.

**Exerciţiul 2.9.6.** Un digraf $D$ poate fi reprezentat şi prin *listele de adiacenţă interioară*, unde D.a[i] este lista vârfurilor-sursă ale arcelor care sosesc în $i$. Să se scrie procedurile care implementează operaţiile tipului Digraf pentru cazul în care digrafurile sunt reprezentate prin listele de adiacenţă interioară.

MATEMATICA

**Exerciţiul 2.9.7.** Să se scrie o procedură `Conex(D : TDigrafListAd) : Boolean` care decide dacă graful reprezentat de `D` este conex sau nu.

**Exerciţiul 2.9.8.** Să se scrie o procedură `Arbore(D : TDigrafListAd) : Boolean` care decide dacă graful reprezentat de `D` este arbore sau nu.
*Indicaţie.* Se poate utiliza faptul că un graf cu $n$ vârfuri este arbore dacă şi numai dacă este conex şi are $n-1$ muchii [Cro92].

**Exerciţiul 2.9.9.** Să se proiecteze o structură de date pentru reprezentarea componentelor conexe ale unui graf şi să se scrie o procedură care, având la intrare reprezentarea unui graf, construieşte componentele conexe ale acestuia.

**Exerciţiul 2.9.10.** Fie $D = (V, A)$ un digraf cu $n$ vârfuri. Un vârf $i$ se numeşte *groapă* (*sink*) dacă pentru orice alt vârf $j \neq i$ există un arc $(j, i) \in A$ şi nu există arc de forma $(i, j)$. Să se scrie o funcţie `Groapa(D, g)` care decide dacă digraful reprezentat de `D` are o groapă sau nu; dacă da, atunci variabila `g` va memora o asemenea groapă. Algoritmul descris de program va avea timpul de execuţie $O(n)$. Pot exista mai multe gropi?

**Exerciţiul 2.9.11.** Se consideră un arbore $G$ (graf conex fără circuite) cu muchiile colorate cu culori dintr-un alfabet $A$. Să se scrie un program care, pentru un şir $a \in A^*$ dat, determină dacă există un drum în $G$ etichetat cu $a$.

**Exerciţiul 2.9.12.** Mulţimea *grafurilor serie-paralel* $(G, s, t)$, unde $G$ este un multigraf (graf cu muchii multiple), iar $s$ şi $t$ sunt vârfuri în $G$ numite *sursă*, respectiv *destinaţie*, este definită recursiv astfel:

- Orice muchie $G = \{u, v\}$ defineşte două grafuri serie-paralel: $(G, u, v)$ şi $(G, v, u)$.
- Dacă $(G_1, s_1, t_1)$ şi $(G_2, s_2, t_2)$ sunt două grafuri serie-paralel, atunci:
  - *compunerea serie* $G_1G_2 = (G, s_1, t_2)$, obţinută prin reuniunea disjunctă a grafurilor $G_1$ şi $G_2$ în care vârfurile $s_2$ şi $t_1$ sunt identificate, este graf serie-paralel;
  - *compunerea paralelă* $G_1 \| G_2 = (G, s_1 = s_2, t_1 = t_2)$, obţinută prin reuniunea disjunctă a grafurilor $G_1$ şi $G_2$ în care sursele $s_1$ şi $s_2$ şi respectiv destinaţiile $t_1$ şi $t_2$ sunt identificate, este graf serie-paralel.

Definiţia este sugerată grafic în figura 2.30.

Să se scrie un program care decide dacă un graf dat este serie-paralel.

**Exerciţiul 2.9.13.** Să se scrie un program care, pentru un graf $G = (V, E)$ şi $i_0 \in V$ date, enumeră toate drumurile maximale care pleacă din $i_0$.

**Exerciţiul 2.9.14.** Se consideră problema din exerciţiul 2.9.13. Presupunem că muchiile grafului $G$ sunt etichetate cu numere întregi. Să se modifice programul de la 2.9.13 astfel încât drumurile să fie enumerate în ordine lexicografică.

**Exerciţiul 2.9.15.** Se numeşte triunghi într-un graf $G$ un ciclu de lungime 3. Să se scrie un program care să determine dacă graful $G$ conţine un triunghi.

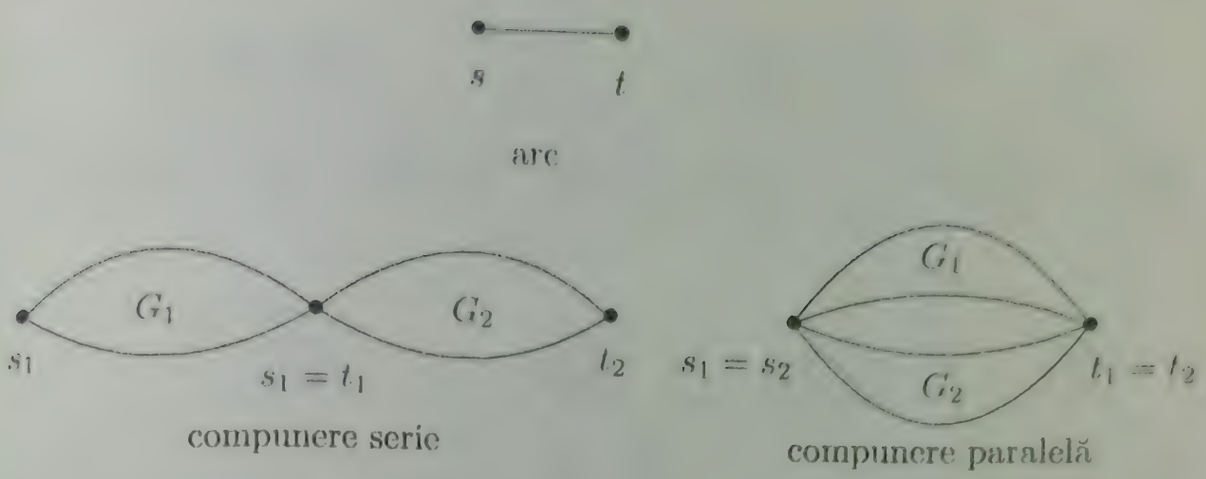



Figura 2.30: Grafuri serie-paralel

## 2.10 *Union-find*

Grafurile pot reprezenta colecţii de mulţimi disjuncte într-un mod foarte natural: vârfurile corespund obiectelor, iar o muchie are semnificaţia că extremităţile sale sunt în aceeaşi mulţime. Această reprezentare permite rezolvarea eficientă a multor probleme legate de mulţimi. De exemplu, a răspunde la întrebarea „Cărei mulţimi îi aparţine obiectul $x$" (operaţia find) presupune de fapt a preciza cărei componente conexe îi a aparţine vârful $x$. Există şi un inconvenient al reprezentării: aceeaşi mulţime poate fi reprezentată prin mai multe grafuri conexe. De aceea, de exemplu, informaţiile oferite de parcurgerile sistematice are o utilitate mai redusă.

Un caz aparte se obţine când se dă un aspect dinamic problemei: la diferite momente, două mulţimi se pot reuni într-una singură (operaţia union). Formularea unei cereri de acest tip este de forma: „reuneşte mulţimea la care aparţine $i$ cu mulţimea la care aparţine $j$". Aceasta operaţie se poate realiza foarte simplu prin adăugarea unei muchii care să unească un vârf din componenta conexă corespunzătoare primei mulţimi cu un vârf din componenta corespunzătoare celei de-a doua mulţimi. Rămâne de stabilit care vârfuri candidează la momentul respectiv pentru unire printr-o muchie. Această alegere va trebui să permită construirea de algoritmi eficienţi pentru ambele operaţii: union şi find. Având în vedere că topologia componentelor conexe nu este importantă, vom alege structura de tip arbore cu rădăcină pentru reprezentarea unei mulţimi. Colecţia de mulţimi este reprezentată de o colecţie de arbori (*o pădure*). Pentru reprezentarea unei păduri vom utiliza un tablou `parinte` cu semnificaţia că *parinte*$[i]$ este predecesorul imediat (părintele) vârfului $i$. Mulţimea univers, peste care se consideră colecţia de mulţimi, este aplicată prin funcţia *index* într-un interval de numere întregi $[0, n-1]$. Astfel, o colecţie $C$ va fi reprezentată de numărul întreg $n$ şi tabloul *parinte*. Dacă $i$ este rădăcina unui arbore, atunci *parinte*$[i] = -1$. Un exemplu de colecţie reprezentată astfel este arătat în figura 2.31a.

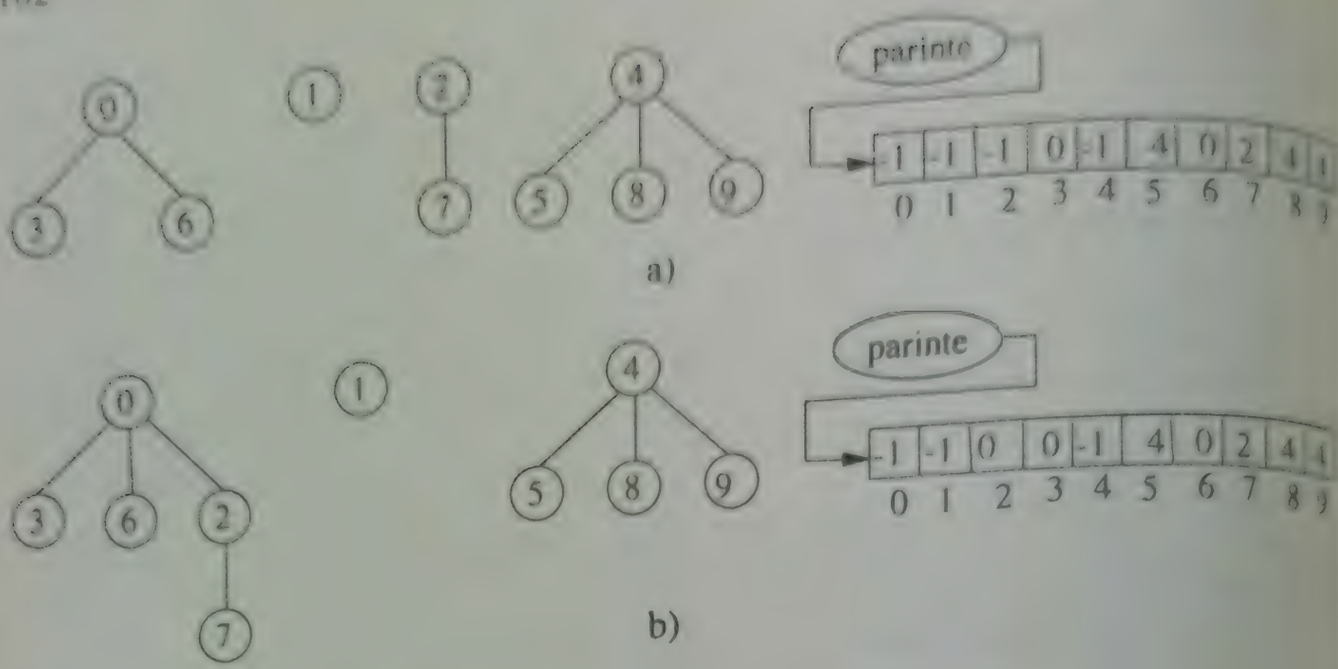



Figura 2.31: Structuri *union-find*

Numele unei mulţimi este dat de rădăcina arborelui ce o reprezintă. Crearea unei mulţimi cu un singur element este banală:

```
procedure singleton(C, i)
   C.parinte[i] ← -1
end
```

Determinarea mulţimii căreia îi aparţine $i$ este echivalentă cu determinarea rădăcinii:

```
function find(C, i)
   temp ← i
   while (C.parinte[temp] ≥ 0) do
      temp ← C.parinte[temp]
   return temp
end
```

Reuniunea a două mulţimi înseamnă construirea unui arc de la rădăcina unui arbore la rădăcina celuilalt:

```
procedure union(C, i, j)
   r1 ← find(i)
   r2 ← find(j)
   if (r1 ≠ r2) then C.parinte[r2] ← r1
end
```

Executând funcţia union(C, 6, 7) pentru colecţia din figura 2.31.a obţinem structura din figura 2.31.b.

Prin execuţia repetată a operaţiei union se pot obţine arbori dezechilibraţi, în care determinarea rădăcinii pentru anumite vârfuri să dureze mult. Structura ar putea fi îmbunătăţită, dacă s-ar putea menţine o formă cât mai aplatizată a arborilor, adică să nu existe noduri aflate la distanţe mari de rădăcină. Aceasta se poate realiza prin memorarea în rădăcini a numărului de vârfuri din arbore (greutatea

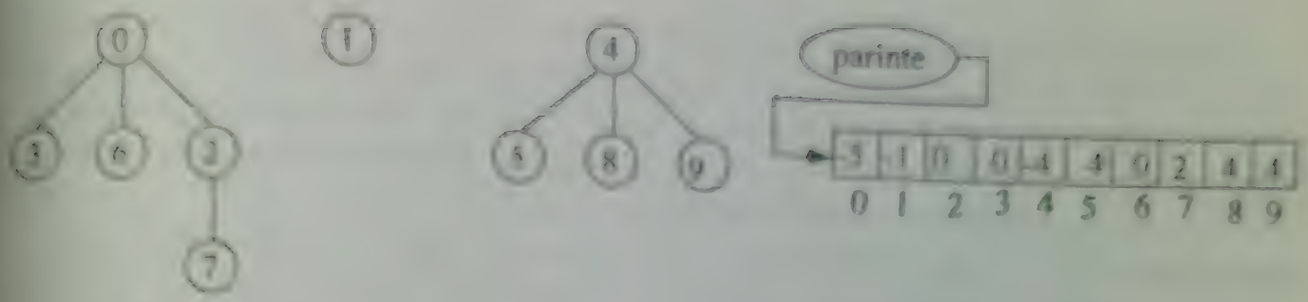



Figura 2.32: Structură "union-find" ponderată

arborelui). Aceasta va fi memorată cu semnul minus pentru a distinge rădăcinile de celelalte noduri. În figura 2.32 este reprezentată o asemenea structură. În plus, procedura union va realiza mai întâi o „aplatizare" a arborilor şi apoi va construi un arc de la rădăcina arborelui cu greutatea mai mare la rădăcina celui cu greutatea mai mică.

```
union(C, i, j)
   r1 ← find(i)
   r2 ← find(j)
   while (C.parinte[i] ≥ 0) do
      temp ← i
      i ← C.parinte[i]
      C.parinte[temp] ← r1
   while (C.parinte[j] ≥ 0) do
      temp ← j
      j ← C.parinte[j]
      C.parinte[temp] ← r2
   if (C.parinte[r1] > C.parinte[r2])
      then C.parinte[r2] ← C.parinte[r1]+C.parinte[r2]
           C.parinte[r1] ← r2
      else if (C.parinte[r1] < C.parinte[r2])
              then C.parinte[r1] ← C.parinte[r1]+C.parinte[r2]
                   C.parinte[r2] ← r1
end
```

O soluţie alternativă la cea de mai sus este să memorăm înălţimea arborelui în loc de greutate. Operaţia union va ancora arborele cu înălţime mai mică de cel cu înălţimea mai mare.

Tipul de date definit mai sus este cunoscut în literatură sub numele de *structura union-find*.

**Teorema 2.3.** *O secvenţă de $m$ operaţii* singleton, union *şi* find *din care $n$ sunt operaţii* singleton *poate fi realizată în timpul $O(m \cdot \log^* n)$ în cazul cel mai nefavorabil, unde $\log^* n$ este cel mai mic număr natural $k$ cu proprietatea $\log^{(k)} n \leq 1$ şi $\log^{(k)}$ desemnează funcţia* log *compusă cu ea însăşi de $k$ ori: $\log^{(0)} n = n$, $\log^{(i+1)} n = \log \log^{(i)} n$.*

Demonstraţia acestei teoreme se reduce la demonstrarea teoremei 2.4.

Ținând cont de componența operației union, aceasta poate fi înlocuită cu două operații find și o operație de legare (link). Operația link se referă la actul legării printr-un arc a celor două rădăcini ale arborilor care reprezintă mulțimile ce se reunesc. Lema următoare arată că această înlocuire nu modifică comportarea asimptotică a unei secvențe de operații singleton, union și find.

**Lema 2.3.** *Dată fiind o secvență $S'$ de $m'$ operații* singleton, union *și* find, *se construiește secvența $S'$ într-o secvență $S$ de $m$ operații* singleton, link *și* find *prin înlocuirea operației* union *cu două operații* find *și o operație* link. *Dacă secvența $S$ are timpul de execuție $O(m \log^* n)$, atunci secvența $S'$ are timpul de execuție $O(m' \log^* n)$.*

*Demonstrație.* O operație union din $S'$ corespunde la trei operații din $S$ (două operații find și o operație link). Rezultă $m' \leq m \leq 3m'$. Aceasta înseamnă că $m = O(m')$. În consecință $O(m \log^* n) = O(m' \log^* n)$. sfdem

Rămâne să demonstrăm că secvența $S$ are timpul de execuție $O(m \log^* n)$.

**Teorema 2.4.** *O secvență $S$ de $m$ operații* singleton, link *și* find, *din care $n$ operații sunt* singleton, *pot fi executate în cazul cel mai nefavorabil în timpul $O(m \log^* n)$.*

Demonstrația acestei teoreme are la baza analiză amortizată și poate fi găsită în [CLR93]

## 2.11 Analiza amortizată

Analiza cazului cel mai nefavorabil poate fi abordată și altfel decât în maniera prezentată în capitolul 1. În locul evaluării izolate a timpului de execuție în cazul cel mai nefavorabil al fiecărei operații asupra unei structuri de date, uneori este benefic să fie evaluat timpul de execuție al unei secvențe de operații sau al tuturor operațiilor asupra structurii de date. Analiza amortizată definește acest proces [CLR00]. Se demonstrează astfel că operațiile compuse din alte operații au cost mediu mic, deși în cazul cel mai nefavorabil, costul este mult mai mare. Aceasta se justifică prin faptul că între operațiile care compun o operație multiplă există interdependențe ce trebuie luate în calcul atunci când se încearcă determinarea unui majorant pentru timpul de execuție al ansamblului (operația multiplă). Analiza amortizată diferă de analiza cazului mediu, deoarece nu sunt implicate probabilități. Analiza amortizată garantează performanța medie a unei operații în cazul cel mai nefavorabil, adică valoarea evaluată a timpului de execuție pentru o secvență de operații asupra unei structuri de date nu este mai mică decât valoarea reală a acestuia.

### 2.11.1 Metoda agregării

Prin această metodă de analiză amortizată se determină costul mediu al unei operații ca fiind $\frac{T(n)}{n}$, unde $T(n)$ este timpul de execuție în cazul cel mai nefavorabil al unei secvențe de $n$ operații. Acest cost va fi numit costul amortizat și va fi aplicat la fiecare operație din secvență.

Pentru exemplificare vom folosi operația de extragere multiplă dintr-o stivă. La operațiile de bază cu stiva vom adăuga operația MultiPop definită astfel:

MultiPop. Elimină ultimele k elemente introduse în stivă.

*Intrare:* o stivă $S$, un număr întreg $k > 0$;
*Ieşire:* stiva $S$ din care s-au eliminat ultimele $k$ elemente introduse, dacă $S$ are cel puţin $k + 1$ elemente,
stiva vidă, dacă $S$ stiva conţine cel mult $k$ elemente.

Dacă stiva este implementată cu structuri dinamice simplu înlănţuite, operaţia MultiPop este implementată de subprogramul `MultiPop(S,k)` de mai jos:

```
procedure MultiPop(S,k)
   while (S ≠ NULL and k ≠ 0) do
      pop(S)
      k ← k-1
end
```

Ne propunem să determinăm timpul de execuţie pentru `MultiPop(S,k)` în cazul unei stive cu $s$ elemente. Numărul de iteraţii ale buclei while este $\min(s, k)$. În fiecare iteraţie se execută o operaţie `pop(S)` şi o atribuire. Rezultă că timpul de execuţie pentru `MultiPop(S,k)` este $O(\min(s, k))$

Să analizăm acum o secvenţă de $n$ operaţii Push, Pop şi MultiPop asupra unei stive $S$ iniţial vide. Pentru simplitatea limbajului, în continuare vom identifica operaţiile Push, Pop şi MultiPop cu subprogramele `Push(S)`, `Pop(S)` şi `MultiPop(S,k)`. Cazul cel mai nefavorabil pentru `MultiPop(S,k)` este dat de stiva cu $n$ elemente şi $k \geq n$. Numărul maxim de elemente ce pot compune stiva $S$ este $n$, deoarece numărul maxim de operaţii `Push(S)` este $n$. Timpul de execuţie al unei operaţii `Push(S)` sau `Pop(S)` este $O(1)$. Rezultă că, în cazul cel mai nefavorabil, timpul de execuţie al operaţiei `MultiPop(S,k)` este $O(n)$. În consecinţă, în acest caz, timpul de execuţie a secvenţei de $n$ operaţii `Push(S)`, `Pop(S)` şi `MultiPop(S,k)` este $O(n^2)$. Deşi analiza este corectă, adunând timpii de execuţie în cazul cel mai nefavorabil pentru fiecare operaţie, se obţine un majorant al timpului de execuţie mult mai mare decât marginea superioară.

Prin analiza amortizată se poate obţine un majorant mai bun. Astfel, deşi o singură operaţie `MultiPop(S,k)` este costisitoare, orice secvenţă de $n$ operaţii `Push(S)`, `Pop(S)` şi `MultiPop(S,k)` asupra unei stive iniţial vide are timpul de execuţie cel mult $O(n)$. Aceasta se explică prin faptul că numărul elementelor eliminate din stivă nu poate depăşi numărul elementelor introduse în stivă. Menţionăm că, iniţial, stiva este vidă.

Numărul operaţiilor `Pop(S)`, inclusiv al celor apelate din `MultiPop(S,k)`, este mai mic sau egal cu numărul operaţiilor `Push(S)`. Rezultă pentru secvenţa celor $n$ operaţii `Push(S)`, `Pop(S)` şi `MultiPop(S,k)` un timp de execuţie de $O(n)$. Costul amortizat pentru o operaţie va fi $\frac{O(n)}{n} = O(1)$. Suma costurilor amortizate (costul mediu pentru cazul cel mai nefavorabil) este evident egală cu majorantul timpului de execuţie obţinut prin metoda agregării. Astfel, prin utilizarea costului mediu este garantat faptul că valoarea evaluată a timpului de execuţie pentru o secvenţă de operaţii asupra unei structuri de date nu este mai mică decât valoarea reală a acestuia.

Am arătat astfel că pentru cazul cel mai nefavorabil, prin metoda agregării, se poate obţine un majorant mai bun pentru timpul de execuţie decât prin analiza

separată a fiecărei operaţii.

### 2.11.2 Metoda conturilor

Prin metoda conturilor, anumite operaţii sunt supraevaluate, iar altele sunt subevaluate. Valoarea astfel asociată unei operaţii va fi numită cost amortizat. Dacă acest cost depăşeşte costul real, diferenţa este memorată drept credit în structura de date. Acest credit va fi folosit mai târziu de operaţiile subevaluate, adică de cele cu costul amortizat inferior costului real. Diferenţa faţă de metoda agregării constă în faptul că acum operaţiile nu mai au acelaşi cost amortizat.

Costul amortizat trebuie ales cu atenţie. Costul amortizat al unei secvenţe de operaţii trebuie să fie un majorant al costului real al secvenţei (timpul real de execuţie). Mai mult, aceasta trebuie să se întâmple pentru orice secvenţă de operaţii. Rezultă că structura de date trebuie să memoreze în orice moment un credit pozitiv, ceea ce asigură faptul că valoarea costului amortizat total este mai mare decât valoarea costului real total.

Pentru a exemplifica metoda conturilor vom folosi acelaşi exemplu ca în cazul metodei agregării. Timpul de execuţie al unei operaţii `Push(S)` sau `Pop(S)` este $O(1)$. Vom considera acest timp ca unitate de cost. Astfel, costul real pentru fiecare din aceste operaţii va fi 1. Timpul de execuţie pentru `MultiPop(S,k)` este $O(\min(s,k))$. Dacă folosim ca unitate de cost timpul de execuţie al unei operaţii `Push(S)` sau `Pop(S)`, atunci costul lui `MultiPop(S,k)` este $\min(s,k)$.

Pentru a asigura un credit permanent pozitiv, vom asocia următoarele costuri amortizate: 2 pentru `Push(S)`, 0 pentru `Pop(S)` şi 0 pentru `MultiPop(S,k)`. Se observă că, pentru `MultiPop(S,k)`, costul este o constantă (0), deşi costul real depinde de dimensiunea stivei $S$ şi de valoarea $k$.

Costul real al operaţiei `Push(S)` este 1, în timp ce costul amortizat este 2. Astfel, operaţia `Push(S)` aplicată unui element va consuma doar 1/2 din costul amortizat asociat acesteia. În consecinţă, fiecare element din stivă va dispune de un credit 1, care va acoperi costul eliminării acestuia din stivă (`Pop(S)`). Operaţia `MultiPop(S,k)` nu necesită cost amortizat strict pozitiv, deoarece este compusă numai din operaţii `Pop(S)` creditate în momentul aplicării operaţiei `Push(S)`. Rezultă că orice secvenţă de operaţii aplicate unei stive va avea un cost amortizat pozitiv.

Costul amortizat al unei secvenţe de operaţii `Push(S)`, `Pop(S)` şi `MultiPop(S,k)` va fi întotdeauna un majorant al costului real. Aceasta este justificată de faptul că elementele aflate la un moment dat în stivă dispun de credit neconsumat, iar operaţiile efectuate până la acel moment au costurile reale acoperite. Costul amortizat este $O(n)$, iar costul real este cel puţin $n$. Rezultă un cost real de $O(n)$.

### 2.11.3 Metoda potenţialului

Metoda potenţialului diferă de metoda conturilor prin faptul că efortul de calcul estimat a fi consumat ulterior nu mai este memorat ca un credit al unui obiect, ci ca energie potenţială a structurii de date, energie care va fi eliberată în momentul execuţiei anumitor operaţii asupra structurii. Metoda poate fi descrisă astfel:

Se presupune că structura inițială este $S_0$ și asupra acesteia urmează a se efectua $n$ operații. Se notează cu $c_i$ costul real al operației cu numărul de ordine $i$ și cu $S_i$ structura rezultată din aplicarea acestei operații asupra structurii $S_{i-1}$. Peste șirul structurilor de date se definește o funcție potențial care asociază fiecărei structuri $S_i$ un număr real $\Pi(S_i)$. Costul amortizat al operației cu numărul de ordine $i$ se definește prin relația:

$$\hat{c}_i = c_i + \Pi(S_i) - \Pi(S_{i-1})$$

Costul amortizat al unei operații este costul real plus diferența de potențial datorată execuției operației. Din relația de mai sus rezultă că pentru $n$ operații costul este:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Pi(S_i) - \Pi(S_{i-1})) = \sum_{i=1}^{n} c_i + \Pi(S_n) - \Pi(S_0)$$

Dacă $S_n \geq S_0$, atunci costul total amortizat este un majorant al costului real. Din nefericire, nu se cunoaște întotdeauna numărul operațiilor ce urmează a fi efectuate asupra unei structuri. Din acest motiv, pentru ca proprietatea de majorant să fie asigurată, este nevoie ca $(\forall i) S_i \geq S_0$.

Intuitiv, dacă $S_i - S_{i-1} > 0$, atunci operația cu numărul de ordine $i$ realizează o încărcare a structurii cu o cantitate de energie potențială, iar dacă $S_i - S_{i-1} < 0$, atunci operația cu numărul de ordine $i$ realizează o descărcare din structura de date a unei cantități de energie potențială. Costul amortizat depinde de modul în care este definită funcția potențial. Este de dorit ca funcția potențial să confere costului amortizat calitatea de margine superioară a costului real.

Vom folosi în continuare exemplul stivei pentru a explica metoda potențialului. Vom defini funcția potențial astfel:

$$\Pi : \{S \mid S = stiva\} \to \mathbb{Z},\ \Pi(S) = \text{numărul elementelor din stivă.}$$

Pentru stiva $S_0$ vom avea $\Pi(S_0) = 0$. Deoarece numărul elementelor dintr-o stivă nu este negativ, stiva $S_i$ rezultată din aplicarea operației cu numărul de ordine $i$ va avea întotdeauna potențial pozitiv, adică $(\forall i) S_i \geq 0 = S_0$. Astfel, costul amortizat pentru $n$ operații este întotdeauna un majorant al costului real.

Să calculăm acum costul amortizat pentru fiecare dintre operațiile `Push(S)`, `Pop(S)` și `MultiPop(S,k)`.

Dacă operația cu numărul de ordine $i$ este `Push(S)`, iar stiva conține $s$ elemente, atunci $\Pi(S_i) - \Pi(S_{i-1}) = (s+1) - s = 1$. Costul amortizat va fi $\hat{c}_i = c_i + \Pi(S_i) - \Pi(S_{i-1}) = 1 + 1 = 2$.

Să presupunem acum că operația cu numărul de ordine $i$ este `Pop(S)`. Diferența de potențial în acest caz va fi $\Pi(S_i) - \Pi(S_{i-1}) = (s-1) - s = -1$, dacă stiva $S$ nu este vidă. Costul amortizat va avea valoare 0, deoarece costul real al operației `Pop(S)` este 1. Dacă stiva $S$ este vidă $\Pi(S_i) - \Pi(S_{i-1}) = 0 - 0 = 0$, costul amortizat al operației `Pop(S)` va fi tot 0, deoarece costul real în acest caz este 0.

Dacă operația cu numărul de ordine $i$ este `MultiPop(S,k)`, atunci diferența de potențial va fi $\Pi(S_i) - \Pi(S_{i-1}) = (s - k') - s' = -k'$, unde $k' = \min(k, s)$

(costul real al operației). Astfel, costul amortizat al operației `MultiPop(S,k)` va fi $\hat{c}_i = c_i + \Pi(S_i) - \Pi(S_{i-1}) = k' - k' = 0$.

Rezultă pentru fiecare dintre operațiile `Push(S)`, `Pop(S)` și `MultiPop(S,k)` un cost de $O(1)$, deci un cost amortizat de $O(n)$ pentru $n$ operații. Din definiția funcției potențial rezultă că $(\forall i) S_i - S_0 \geq 0$, ceea ce implică pentru costul amortizat al celor $n$ operații proprietatea de majorant pentru costul real. În cazul cel mai nefavorabil, costul celor $n$ operații este astfel $O(n)$.

## 2.12 Referințe bibliografice

Tipurile de date de nivel înalt sunt analizate intens în literatura de specialitate [CLR93, CLR00, Sah98, Baa78, BG00, Mel84a, AHU74, HS84, MB00, Wei92, LG86, CB02]. Analiza amortizată este prezentată în detaliu în [CLR93, CLR00, Hei96].

# Capitolul 3

# Derecursivare

În acest capitol prezentăm câteva strategii prin care soluţiile recursive pot fi descrise prin programe nerecursive. Dintre toate motivele care impun necesitatea cunoaşterii unor asemenea strategii, următoarele două ni se par a fi cele mai importante:

- Uneori, eficienţa algoritmilor descrişi prin programe nerecursive este mult mai bună decât a celor descrişi recursiv. Vom vedea cazuri când descrierea recursivă are complexitate exponenţială, iar descrierea nerecursivă complexitate liniară.
- Execuţia subprogramelor recursive presupune gestionarea (cel puţin a) unei stive. Prin derecursivare, programatorul îşi gestionează singur această stivă (în cazul programelor recursive aceasta este gestionată de sistem) şi poate astfel exploata informaţia conţinută în stivă pentru a creşte calitatea sau gradul de utilizare a programului.

Tehnicile descrise în această secţiune se referă numai la recursia directă.

## 3.1 Parcurgerea în inordine a arborilor binari

Sunt multe cazuri în activitatea de programare când prelucrarea informaţiei dintr-un nod al unui arbore binar este însoţită de o procesare a informaţiei aflate pe drumul de la rădăcina arborelui la nod. Vom vedea că, prin derecursivare, acest drum este memorat chiar de stiva cu valorile parametrilor.

Reamintim mai întâi varianta recursivă a algoritmului de parcurgere în inordine a arborilor binari:

```
procedure inordine(v)
1:  if v ≠ NULL
2:      then inordine(v->stg)
3:           viziteaza(v)
4:           inordine(v->drp)
end
```

Tehnica utilizată este inspirată din descrierea semanticii operaţionale a instrucţiunii de apelare a unui subprogram. Vom descrie un program nerecursiv care simulează

activitatea celui recursiv. Programul cu cea mai simplă structură este obţinut în urma unui proces de rafinare.

**Varianta 1.** Prima variantă utilizează cele două stive de la descrierea semanticii operaţionale a instrucţiunii de apelare a unui subprogram:

- o stivă pentru parametrii în care se vor memora referinţe de noduri, notată `spar`;
- o stivă de protocol în care sunt memorate adresele de întoarcere în subprogram, notată `sadr`. Adresele sunt identificate prin intermediul etichetelor.

În general, fiecare apel recursiv este simulat de o secvenţă de instrucţiuni care realizează:

- salvarea stării curente a subprogramului. Aceasta constă din:
  - introducerea valorilor (în cazul apelului prin valoare) sau adreselor (în cazul apelului prin referinţă) parametrilor din instrucţiunea de apelare în stiva parametrilor; dacă există variabile locale, atunci se salvează în stivă şi valorile acestora,
  - introducerea adresei primei instrucţiuni ce urmează apelului în stiva adreselor de retur;
- actualizarea valorile parametrilor pentru noul apel recursiv;
- saltul la prima instrucţiune din subprogram.

Astfel, primul apel recursiv de la eticheta 2 se înlocuieşte cu:

```
push(spar, v)  //introduce radacina subarborelui in
               //                 stiva de parametri
push(sadr, 3)  //introduce adresa de retur in stiva
               //                       de protocol
v ← v->lstg;   //inainteza in subarborele aflat la
               //                             stanga
goto 1         //se reia calculul procedurii pentru
               //                 subarborele stang
```

Analog, al doilea apel recursiv va fi înlocuit cu:

```
push(spar, v)
push(sadr, 5)
v ← v->ldrp
goto 1
```

Simularea terminării execuţiei unui apel necesită următoarele operaţii:

- Se restabileşte starea subprogramului de dinaintea ultimului apel recursiv. Aici, aceasta constă din extragerea din stiva de parametri a valorilor corespunzând rădăcinii subarborelui care tocmai s-a vizitat. În general, sunt extraşi toţi parametrii corespunzători apelului care tocmai s-a terminat şi se actualizează parametrii apelului suspendat (ce urmează a se relua). Dacă există variabile locale, atunci şi acestea vor fi supuse unui proces de extragere şi actualizare.
- Se reia calculul cu instrucţiunea dată de vârful stivei de protocol; această adresă va fi eliminată din stivă.

Secvenţa de instrucţiuni care realizează aceste operaţii este:

```
if not esteVida(spar)
  then v ← top(spar); //radacina arborelui ce s-a terminat
       pop(spar)      //de vizitat este eliminata din stiva
       adret ← top(sadr); //adresa de retur de la care se
                      //                  continua calculul
  pop(sadr);
  switch (adret)      //salt la instructiunea data de
                      //adresa de retur
    case 3
      goto 3
    case 5
      goto 5
```

Asamblând secvenţele de mai sus obţinem următorul subprogram nerecursiv:

```
procedure inordNerec1(t)
  // initializari
  spar ← stivaVida()
  sadr ← stivaVida()
  v ← t
1:if v ≠ nil
     then // simuleaza primul apel recursiv
2:        push(spar, v)
          push(sadr, 3)
          v ← v->lstg;
          goto 1
3:viziteaza(v)
  //simuleaza al doilea apel recursiv
4:push(spar,v)
  push(sadr,5)
  v ← v->ldrp
  goto 1
  //simuleaza intoarcerea
5:if not esteVida(spar)
     then v ← Top(spar)
          pop(spar)
          adret ← top(sadr)
          pop(sadr)
    switch (adret)
      case 3
        goto 3
      case 5
        goto 5
end
```

**Varianta 2.** Se obţine prin structurarea primei variante. Un vârf $v$, după ce a fost vizitat, este introdus în stiva parametrilor prin instrucţiunea de la eticheta 4. Când se execută instrucţiunea de la eticheta 5 şi în vârful stivei se află $v$, adresa de retur fiind 5, $v$ este eliminat din stivă fără ca asupra lui să se efectueze vreo operaţie. De aici rezultă că introducerea sa în stivă este inutilă. Astfel că putem pune secvenţele de la etichetele 3 şi 4 la un loc direct în secvenţa de la 5.

```
procedure inordNerec2(t)
  spar ← stivaVida()
  sadr ← stivaVida()
  v ← t
  repeat
    while v ≠ nil do
      push(spar, v)
      push(sadr, 3)
      v ← v->lstg
    if not esteVida(spar)
       then v ← top(spar)
            pop(spar)
            adret ← top(sadr)
            pop(sadr)
            switch (adret)
              case 3
                viziteaza(v)
                v ← v->ldrp
  until esteVida(spar) and (v = NULL)
end
```

**Varianta 3.** Se obţine din varianta 2 prin eliminarea stivei de protocol sadr, a cărei prezenţă este acum inutilă.

```
procedure inordNerec3(t)
  spar ← stivaVida()
  v ← t
  repeat
    while v ≠ NULL do
      push(spar, v)
      v ← v->lstg
    if (not esteVida(spar))
       then v ← top(spar)
            pop(spar)
            viziteaza(v)
            v ← v->ldrp
  until esteVida(spar) and (v = NULL)
end
```

## 3.2 Recursia liniară

O schemă de recursie liniară este o definiţie de forma:

$$f(x) = \textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } \varphi(k(x), f(h(x))) \textbf{ fi}$$

Un exemplu de funcţie definită prin recursie liniară este factorialul: luăm $p(x) \equiv (x \leq 1)$, $k(x) = x$, $h(x) = x - 1$ şi $\varphi(x, y) = x \cdot y$. Un calcul sintactic al unei funcţii definite prin recursie liniară arată ca în figura 3.1. Acest calcul este finit dacă şi numai dacă există un $n$ natural cu proprietatea $p(h^n(a)) = true$; în acest caz, calculul se opreşte la cel mai mic număr natural nenul $n$ cu această proprietate.

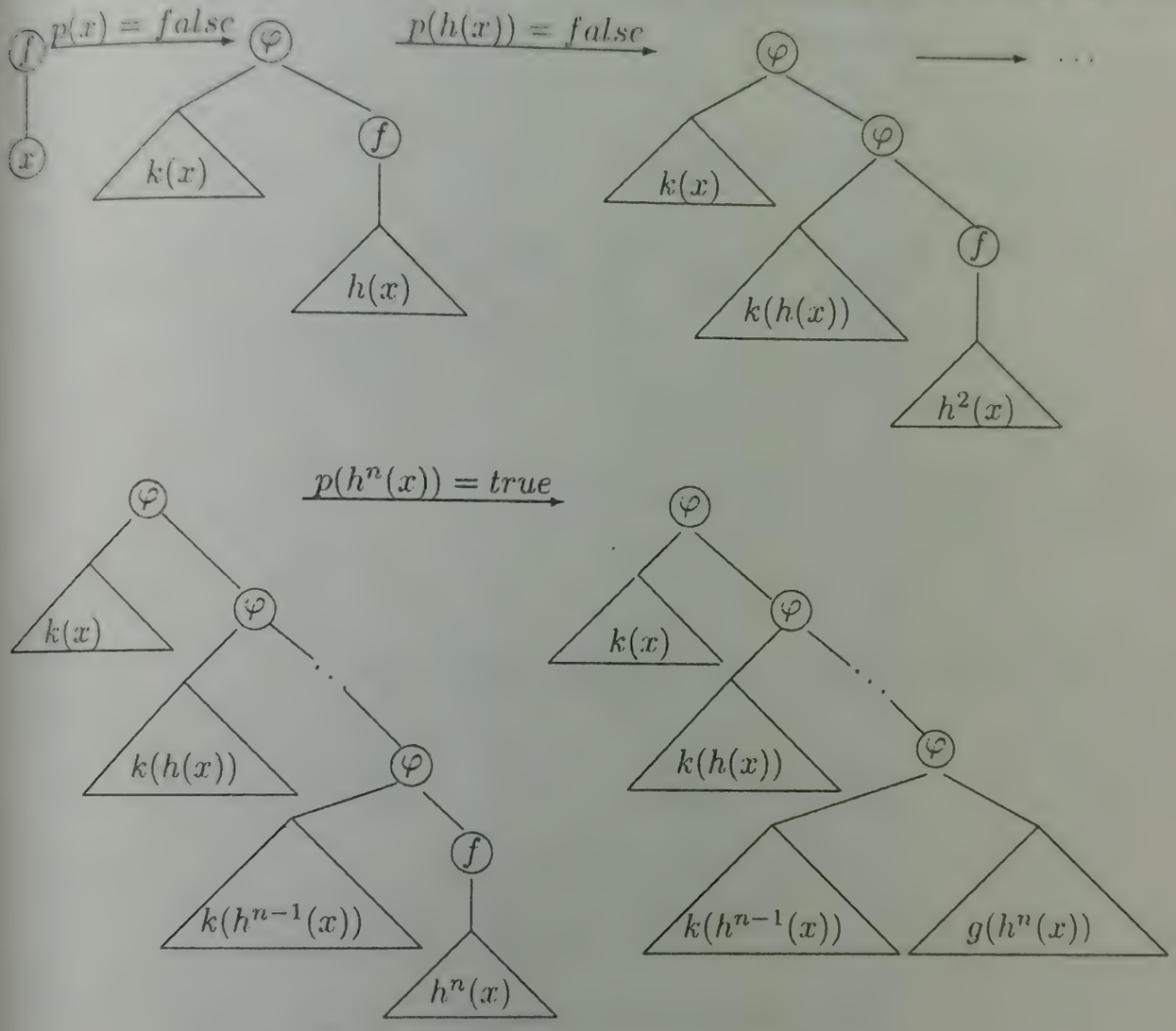



Figura 3.1: Calculul sintactic al recursiei liniare

Programul nerecursiv, care simulează calculul apelurilor recursive, se obţine în modul următor:

- se utilizează o stivă în care se vor memora valorile parametrilor din apelurile recursive: $a$, $h(a)$, $h^2(a)$), ... Stiva cu adresele de retur nu mai este necesară. Se observă că prin introducerea parametrilor în stivă se simulează, de fapt, construcţia arborelui final din calculul sintactic;

- după terminarea unui apel, cu ajutorul valorilor din vârful stivei, se evaluează funcţia $\varphi$. Procedeul continuă până când stiva devine vidă (după utilizarea sa la calculul funcţiei $\varphi$, elementul corespunzător este eliminat din stivă). În reprezentarea arborescentă, această fază simulează evaluarea arborelui final din calculul sintactic.

Strategia de mai sus este descrisă de următorul program nerecursiv:

```
function f(x)
  // initializare
  u ← x
  s ← stivaVida()
  // constructia arborelui
  while not p(u) do
    push(S, u)
    u ← h(u)
  // evaluarea arborelui
  v ← g(u)
  while not EsteStivaVida(S) do
    u ← top(S); pop(S)   // extrage parametrul din stiva
    w ← k(u)             // evaluarea functiei k
    v ← phi(w,v)         // evaluarea functiei phi
  return v
end
```

Rolul variabilelor `u, v, w` este sugerat grafic în figura 3.2.

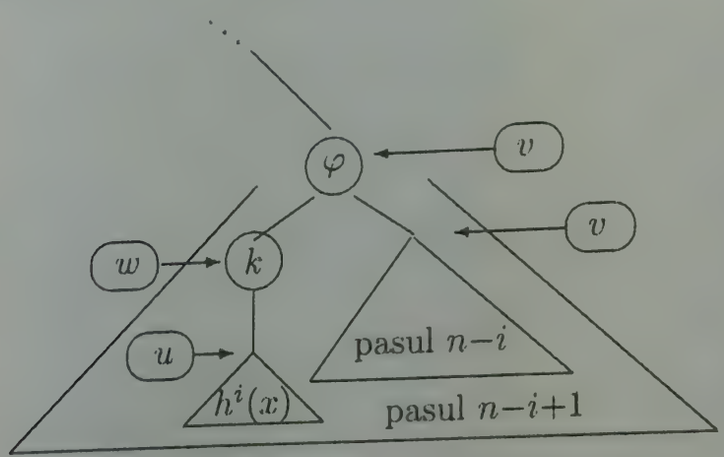



Figura 3.2: Rolul variabilelor `u, v, w`

Există cazuri când programul de mai sus poate fi simplificat foarte mult. Iată două dintre acestea:

1. În cazul când $h$ admite inversă, se poate renunţa la utilizarea stivei. Se utilizează un contor şi introducerea în stivă se înlocuieşte cu incrementarea contorului, iar extragerea din stivă este înlocuită cu instrucţiuni care realizează $u \leftarrow h^{-1}(u)$ şi decrementarea contorului. Rolul primei faze, corespunzătoare construcţiei arborelui şi a contorului, este de a-l determina pe cel mai mic $n$ cu $p(h^n(x)) = true$. De exemplu, pentru factorial, avem $h^{-1}(x) = x + 1$.

2. Dacă există o funcţie care determină direct cel mai mic $n$ cu proprietatea $p(h^n(x)) = true$, atunci instrucţiunea `while` poate fi înlocuită cu o instrucţiune `for`. De asemenea, faza corespunzătoare construcţiei arborelui nu mai este necesară. În cazul funcţiei factorial, avem $n = x - 1$.

Prezentăm în continuare cele trei variante care conduc la forma simplificată a programului nerecursiv ce calculează factorialul.

**Varianta 1.** Corespunde schemei generale cu utilizarea stivei.

```
function fact(x)
  //initializare
  u ← x
  s ← stivaVida()
  //constructia arborelui
  while (u > 1) do
    push(s, u)
    u ← u-1
  //evaluarea arborelui
  v ← 1
  while not esteVida(s) do
    u ← top(s)
    pop(s)
    w ← u
    v ← w*v
  return v
end
```

**Varianta 2.** Această variantă este obţinută din prima prin implementarea stivei cu ajutorul unui simplu contor.

```
function fact(x)
  //initializare
  u ← x
  contor ← 0
  //determinarea lui n
  u ← x
  if (u = 0) then u ← 1
  while (u > 1) do
    contor ← contor + 1
    u ← u-1
  v ← 1
  while (contor > 0) do
    contor ← contor - 1
    u ← u+1
    w ← u
    v ← w * v
  return v
```

```
end
```

**Varianta 3.** Deoarece se cunoaşte $n = r-1$, prima fază corespunzătoare construcţiei arborelui poate fi eliminată.

```
function fact(x)
  //initializare
  v ← 1
  //proces iterativ
  for u ← x downto 1 do
    v ← u * v
  return v
end
```

## 3.3 Recursia în cascadă

Schema de recursie în cascadă are următoarea definiţie:

$$f(x) = \textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } \varphi(f(k(x)), f(h(x))) \textbf{ fi}$$

Un exemplu de funcţie definită prin recursie în cascadă îl constituie funcţia Fibonacci. Considerăm mai întâi subprogramul recursiv care calculează o funcţie definită prin recursie în cascadă:

```
function f(x)
1: if p(x)
2:    then f ← g(x)
3:    else
4:         u ← k(x)
5:         v1 ← f(u)
6:         u ← h(x)
7:         v2 ← f(u)
8:         return phi(v1, v2)
end;
```

Procedura de mai sus utilizează mai multe variabile locale decât ar fi necesare, dar ele vor face prezentarea mai explicită. Putem descrie un program nerecursiv, care îl simulează pe cel de mai sus, procedând ca la parcurgerea în inordine a arborilor binari. Pentru memorarea parametrilor utilizăm două stive: o stivă *spar* care memorează argumentele pentru apelurile funcţiei $f$ şi o stivă *srez*, unde sunt memorate rezultatele evaluărilor parţiale ale funcţiei $f$. Stiva *spar* poate fi utilizată şi pentru salvarea eventualelor variabile locale.

```
function  f(x)
   //initializari
   spar ← stivaVida()
   sadr ← stivaVida()
   srez ← stivaVida()
   u ← x
```

```
   //proces iterativ
1: if (p(u))
      then v ← g(u)
           push(srez, v)
      else //simuleaza primul apel
           push(spar, u)
           push(sadr, 6)
           u ← k(u)
           goto 1
6:    //simuleaza al doilea apel
      push(spar, u)
      push(sadr, 8)
      u ← h(u)
      goto 1
8:    //evalueaza phi
      v2 ← top(srez);  pop(srez)
      v1 ← top(srez);  pop(srez)
      v ← phi(v1, v2)
      push(srez,v)
   if (not esteVida(spar))
      then adr ← Top(sadr);   pop(sadr)
           u ← top(spar);     pop(spar)
           switch (adr)
             case 6
                goto 6;
             case 8
                goto 8
      else return top(srez)
end
```

Dacă efectuăm un calcul sintactic (figura 3.3), atunci rezultatul final este un arbore binar în care fiecare nod sau are exact doi subarbori nevizi sau are ambii subarbori vizi. Procedura descrie o strategie de explorare a acestui arbore. Ca şi în cazul parcurgerii de arbori, programul poate fi simplificat. De exemplu, prin structurare, stiva cu adresele de retur poate fi eliminată. Considerăm că efectuarea acestor modificări constituie un bun exerciţiu pentru cititor.

### 3.3.1 Studiu de caz: funcţia Fibonacci

Un subprogram recursiv care calculează funcţia Fibonacci este următorul:

```
function fib(n)
  if (n ≤ 1
    return n
  else
    return fib(n-1) + fib(n-2)
end
```

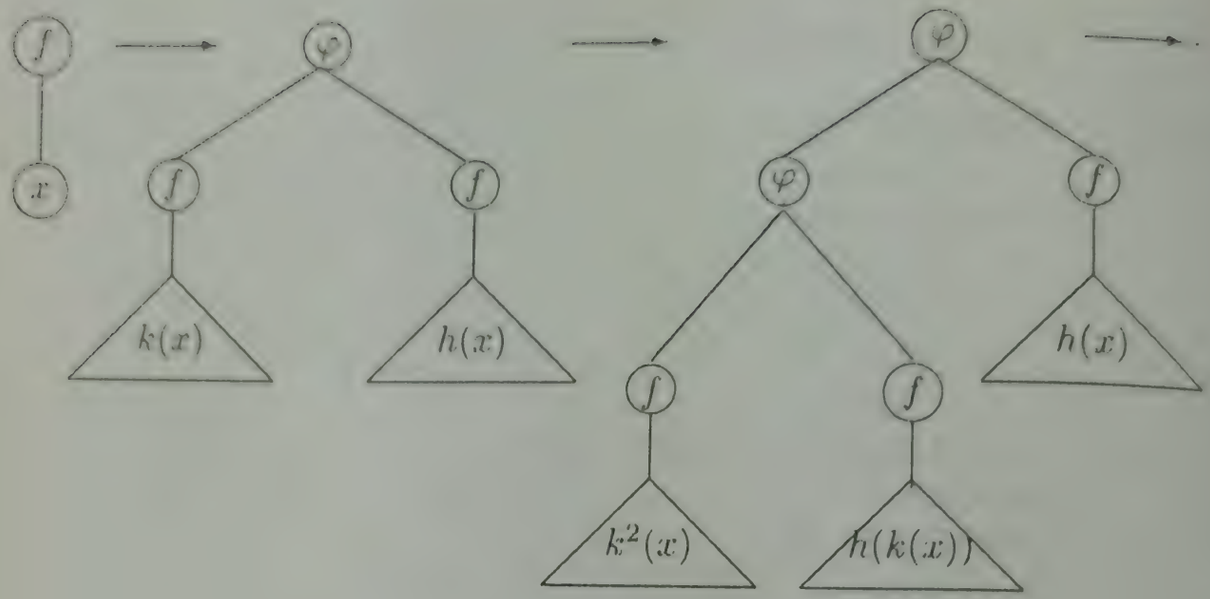



Figura 3.3: Recursie în cascadă

Dacă notăm cu $T(n)$ complexitatea timp a programului de mai sus, rezultă că $T(n)$ satisface relaţia de recurenţă:

$$T(n) = \begin{cases} 1 & \text{, dacă } n \leq 1 \\ T(n-1) + T(n-2) + 1 & \text{, dacă } n > 1 \end{cases}$$

Avem $T(n) > T(n-1) + T(n-2)$, care implică $T(n) > f_n$, unde $f_n$ este cel de-al $n$-lea termen din şirul lui Fibonacci ($f_n = fib(n)$). Ştim că $f_n$ poate fi calculat şi cu formula:

$$f_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right] = O(\phi^n)$$

unde[1] $\phi = \frac{1+\sqrt{5}}{2}$. Deci $f_n$ are o creştere foarte rapidă, exponenţială, şi de aici rezultă că programul recursiv are complexitatea timp exponenţială. Vom arăta că prin derecursivare se poate obţine un program cu complexitate liniară. Considerăm primii paşi din calculul sintactic al funcţiei (figura 3.4). Observăm că $fib(n-2)$ este evaluat de două ori. În varianta nerecursivă vom forţa ca $fib(n-2)$ să fie evaluat o singură dată. Programul nerecursiv va descrie un proces iterativ, la fiecare iteraţie determinându-se o valoare parţială a funcţiei $fib$ (un termen al şirului Fibonacci). Generalizăm, considerând în loc de $n$ o variabilă liberă $i$. Utilizăm trei variabile, $u, v, w$, pentru care vor păstra următoarele relaţii invariante: $u$ memorează $f_{i-1} = fib(i-1)$, $v$ memorează $f_{i-2} = fib(i-2)$ şi $w$ memorează $f_i = fib(i)$.

[1] Numărul $\frac{1+\sqrt{5}}{2}$ este cunoscut sub numele de „numărul de aur" (*golden ratio*), el fiind important atât în diferite ramuri ale matematicii, cât şi în lumea artei. El a fost notat cu litera grecească $\phi$ în memoria lui Phidias, despre care se spune că a utilizat acest număr la realizarea sculpturilor sale.

Menţinerea acestor relaţii invariante se realizează prin următoarele instrucţiuni:

$\{u = f_{i-1} \wedge v = f_{i-2} \wedge w = f_i\}$
v ← u;
$\{u = f_{i-1} \wedge v = f_{i-1} \wedge w = f_i\}$
u ← w;
$\{u = f_i \wedge v = f_{i-1} \wedge w = f_i\}$
w ← u + v;
$\{u = f_i \wedge v = f_{i-1} \wedge w = f_{i+1}\}$
i ← i + 1;
$\{u = f_{i-1} \wedge v = f_{i-2} \wedge w = f_i\}$

Acum, programul nerecursiv poate fi descris astfel:

```
function fib(n)
  //initializari
  u ← 1;        //u = f_1
  v ← 0;        //v = f_0
  w ← 1;        //w = f_2
  i ← 2;
  //proces iterativ
  //invariant: u=f_(i-1), v=f_(i-2), w=f_i
  while (i < n) do
    v ← u
    u ← w
    w ← u + v
    i ← i + 1
  //i = n and w =f_n
  return w
end
```

Se observă imediat că programul nerecursiv are complexitatea timp $O(n)$.

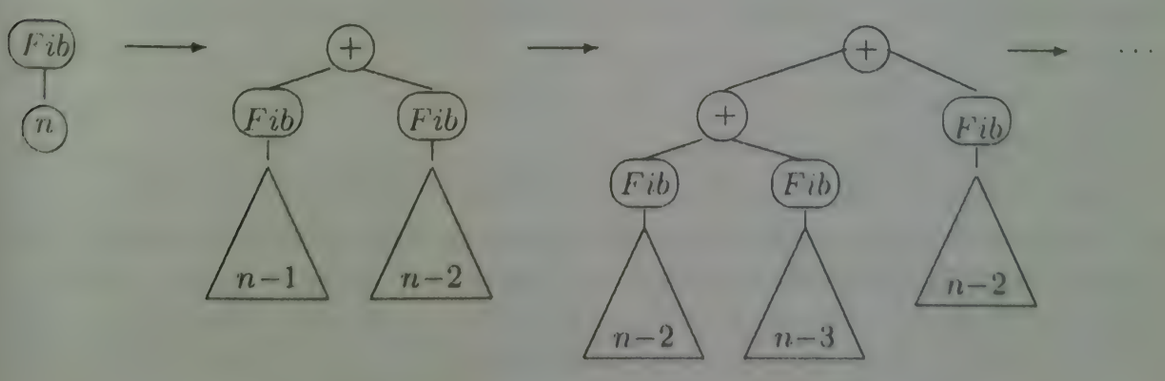



Figura 3.4: Funcţia Fibonacci

## 3.4 Memorarea într-un tabel a rezultatelor subproblemelor

În această subsecţiune vom prezenta o tehnică prin care putem extrage algoritmi nerecursivi de complexitate polinomială (uneori chiar liniară) din algoritmi recursivi cu complexitate exponenţială. Ideea principală a acestei tehnici este următoarea: *se rezolvă subproblemele de la mic la mare şi se memorează rezultatele acestora într-un tabel.* Avantajul acestei strategii constă în faptul că, dacă o subproblemă apare în rezolvarea mai multor subprobleme mai mari, ea va fi rezolvată o singură dată şi valoarea soluţiei sale va fi regăsită în tabel de fiecare dată în timpul $O(1)$. Asupra acestei strategii vom mai reveni în capitolul dedicat programării dinamice. Exemplificăm aplicarea acestei tehnici prin câteva studii de caz.

### 3.4.1 Funcţia Fibonacci

Este exemplul cel mai simplu şi mai elocvent. Am văzut deja că algoritmul descris de subprogramul recursiv are complexitate exponenţială. Presupunem acum că valorile funcţiei *fib* sunt calculate de la mic la mare , *i.e.*, în ordinea $fib(1), fib(2), fib(3), \ldots$ şi că sunt memorate într-un tabel unidimensional `f`. Calculul lui $fib(i)$, $i > 2$, se poate face în timpul $O(1)$ printr-o instrucţiune de forma:

```
f[i] ← f[i-2] + f[i-1]
```

Rezultă că primele $n$ numere Fibonacci pot fi calculate în timpul $O(n)$.

### 3.4.2 Numărul de expresii cu paranteze corect formate

Considerăm expresii scrise numai cu operatori binari. O astfel de expresie este următoarea $((a * b) + c) + (a - b)$. Prin ştergerea operanzilor şi operatorilor obţinem o expresie formată numai cu paranteze: (())(). Expresiile obţinute în acest mod constituie expresiile cu paranteze corect formate. Pentru un număr natural $n$ dat, notăm cu $P(n)$ numărul de expresii cu paranteze corect formate de lungime $n$. Ne propunem găsirea unei relaţii de recurenţă pentru $P(n)$ şi a unui algoritm eficient de calcul al lui $P(n)$.

Mai întâi vom căuta o definiţie recursivă pentru aceste expresii. Notăm cu $E$ mulţimea expresiilor. O definiţie recursivă pentru $E$ este următoarea:

i) $() \in E$;
ii) dacă $e \in E$, atunci $(e) \in E$;
iii) dacă $e_1, e_2$, atunci $e_1 e_2 \in E$ ;
iv) $E$ este cea mai mică mulţime care satisface condiţiile de mai sus.

Un prim fapt care se observă imediat este că orice expresie este de lungime pară (se demonstrează prin inducţie). De aici rezultă că $P(n) = 0$ pentru orice $n$ impar. Rămâne de determinat $P(n)$, pentru $n$ par. Evident, $P(2) = 1$. Fie $n > 2$. Vom încerca să găsim câte descompuneri ii) sau iii) pot exista pentru o expresie de lungime $n$. Se observă uşor că există $P(n-2)$ descompuneri ii). Pentru descompunerile iii) vom vedea mai întâi care sunt valorile posibile pentru lungimea lui $e_1$ (evident, $|e_2| = n - |e_1|$). Nu este greu de observat că aceste valori sunt

$2, 4, \ldots, n-2$. Numărul expresiilor $e_1$ de lungime $i$ este $P(i)$, iar pentru un $e_1$ fixat, numărul expresii de forma iii) este $P(n-i)$. De aici, rezultă următoarea formulă recurentă pentru $P(n)$:

$$P(n) = P(n-2) + \sum_{i \in I(n)} P(i) \cdot P(n-i)$$

unde $I(n) = \{2, 4, \ldots, n-2\}$. Un subprogram recursiv care calculează $P(n)$ are complexitatea timp exponenţială. De exemplu, un apel pentru calculul lui $P(4)$ necesită trei apeluri recursive pentru calculul lui $P(2)$. Calculând valorile $P(n)$ în ordinea $P(2), P(4), \ldots$ şi memorându-le într-un tabel unidimensional obţinem un algoritm de complexitate timp $O(n^2)$.

### 3.4.3 Numărul de partiţii ale unui număr natural

Un număr natural $n$ poate fi scris ca sumă de numere naturale nenule în mai multe moduri (nu vom face distincţie între două sume în care diferă numai ordinea termenilor). De exemplu, 4 poate fi scris ca: 1+3, 1+1+2, 1+1+1+1, 2+2. Notăm cu $S(n)$ numărul de posibilităţi în care $n$ poate fi scris ca sume. Pentru a găsi numărul $S(n)$, considerăm mai întâi câteva cazuri particulare:

$$\begin{aligned} 2 &= 1+1 \\ 3 &= 1+2 \\ &\quad 1+1+1 \\ 4 &= 1+3 \\ &\quad 1+1+2 \\ &\quad 1+1+1+1 \\ &\quad 2+2 \end{aligned}$$

Privind aceste exemple observăm următoarele:

- O sumă poate fi reprezentată unic de o secvenţă de forma $(n_1, n_2, \ldots . n_k)$ cu proprietăţile: $n_1 \leq n_2 \leq \cdots \leq n_k$ şi $n_1 + n_2 + \ldots n_k = n$.
- Sumele lui 3 (care toate încep cu 1) intră în sumele lui 4 care încep cu 1. Deci este util să considerăm numărul $s(i, j)$ al sumelor lui $i + j$ care încep cu $i$. Invităm cititorul să mai considere câteva exemple particulare ($n = 4, 5, 6, 7$) pentru a vedea utilitatea acestei notaţii.

Ne propunem acum să găsim o formulă recurentă pentru $s(i, j)$. Dacă $j < i$, atunci $s(i, j) = 0$ şi dacă $i = j$, atunci $s(i, i) = 1$. În continuare presupunem $i < j$. O sumă a lui $i + j$ care începe cu $i$ este de forma $(i, j)$ sau de forma $(i, k, \ldots)$, unde $(k, \ldots)$ este o sumă al lui $j$ care începe cu $k$. Valoarea minimă pentru $k$ este $i$, iar cea maximă este $\frac{j}{2}$. De aici rezultă următoarea formulă pentru $s(i, j)$ (cazul $i < j$):

$$s(i, j) = \sum_{i \leq k \leq \frac{j}{2}} (1 + s(k, j-k))$$

Valorile $s(i,j)$ pot fi memorate într-un tabel bidimensional. Elementele din acest tabel pot fi calculate în ordinea parcurgerii tabelului pe coloane. Cunoscând valorile $s(i,j)$, numărul $S(n)$ poate fi calculat prin formula:

$$S(n) = \sum_{1 \le i \le \frac{n}{2}} s(i, n-i)$$

Rezultă un algoritm nerecursiv care determină $S(n)$ în timpul $O(n^3)$.

## 3.5 Referinţe bibliografice

Exemple privind proiectarea de algoritmi recursivi pot fi găsite în [Woo84, HE87, HS84]. În [HE87] este explicată foarte bine evoluţia stivei "run-time" în timpul apelurilor recursive. Pentru aspectele teoretice privind recursia se poate consulta [Wan80, Gri81, AA78, Coh90].

# Capitolul 4

# Sortare internă

Sortarea este una dintre problemele cele mai importante atât din punct de vedere practic, cât şi teoretic. Sortarea poate fi formulată în diferite moduri. Cea mai generală formulare şi mai des utilizată este următoarea:

> Fie dată o secvenţă $(v_0, \ldots, v_{n-1})$ cu componentele $v_i$ dintr-o mulţime total ordonată. Problema sortării constă în determinarea unei permutări $\pi$ astfel încât $v_{\pi(0)} \leq v_{\pi(1)} \leq \cdots \leq v_{\pi(n-1)}$ şi în rearanjarea elementelor din secvenţă în ordinea dată de permutare.

O altă formulare, echivalentă cu cea de mai sus, este următoarea:

> Fie dată o secvenţă de structuri statice $(R_0, \ldots, R_{n-1})$, unde fiecare structură $R_i$ are o valoare cheie $K_i$. Peste mulţimea cheilor $K_i$ este definită o relaţie de ordine totală. Problema sortării constă în determinarea unei permutări $\pi$ cu proprietatea $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$ şi în rearanjarea structurilor în ordinea $(R_{\pi(0)}, \ldots, R_{\pi(n-1)})$.

Pentru ambele formulări presupunem că secvenţa dată este reprezentată printr-o listă liniară. Dacă această listă este memorată în memoria internă a calculatorului, atunci avem *sortare internă* şi, dacă se găseşte într-un fişier memorat pe un suport extern, atunci avem *sortare externă*. În acest capitol ne ocupăm numai de sortarea internă.

Vom simplifica formularea problemei presupunând că secvenţa dată este un tablou unidimensional. Acum problema sortării se reformulează astfel:

> *Intrare:* $n$ şi tabloul $(\mathtt{a}[i] \mid i = 0, \ldots, n-1)$ cu $a[i] = v_i$, $i = 0, \ldots, n-1$.
> *Ieşire:* tabloul a cu proprietăţile: $a[i] = w_i$ pentru $i = 0, \ldots, n-1$, $w_0 \leq \cdots \leq w_{n-1}$ şi $(w_0, \ldots, w_{n-1})$ este o permutare a secvenţei $(v_0, \ldots, v_{n-1})$; convenim să notăm această proprietate prin $(w_0, \ldots, w_{n-1}) = \mathrm{Perm}(v_0, \ldots, v_{n-1})$.

Un algoritm de sortare este stabil dacă păstrează ordinea relativă iniţială a elementelor egale.

Există foarte mulţi algoritmi care rezolvă problema sortării interne. Nu este în intenţia noastră a-i trece în revistă pe toţi. Vom prezenta numai câteva metode pe care le considerăm cele mai semnificative. Două dintre acestea, anume sortarea prin interclasare şi sortarea rapidă, vor fi prezentate în capitolul dedicat metodei *divide-et-impera*.

## 4.1 Sortare bazată pe comparaţii

În această secţiune am grupat metodele de sortare bazate pe următoarea tehnică: determinarea permutării se face comparând la fiecare moment două elemente $a[i]$ şi $a[j]$ ale tabloului supus sortării. Scopul comparării poate fi diferit: pentru a rearanja valorile celor două componente în ordinea firească (sortare prin interschimbare), sau pentru a insera una dintre cele două valori într-o subsecvenţă ordonată deja (sortare prin inserţie), sau pentru a selecta o valoare ce va fi pusă pe poziţia sa finală (sortare prin selecţie). Decizia că o anumită metodă aparţine uneia dintre subclasele de mai sus are un anumit grad de subiectivitate. De exemplu, selecţia naivă ar putea fi foarte bine considerată o metodă bazată pe interschimbare.

### 4.1.1 Sortarea prin interschimbare

Vom prezenta întâi strategia cunoscuta sub numele de *sortare prin metoda bulelor* (*bubble-sort*).

Notăm cu $SORT(\mathtt{a})$ predicatul care ia valoarea *true* dacă şi numai dacă tabloul a este sortat. Metoda *bubble-sort* se bazează pe următoarea definiţie a predicatului $SORT(\mathtt{a})$:

$$SORT(\mathtt{a}) \iff (\forall i)(0 \leq i < n-1) \Rightarrow a[i] \leq a[i+1]$$

O pereche $(i, j)$, cu $i < j$, formează o *inversiune (inversare)*, dacă $a[i] > a[j]$. Astfel, pe baza definiţiei de mai sus vom spune că tabloul a este sortat dacă şi numai dacă nu există nici o inversiune $(i, i+1)$. Metoda *bubble-sort* propune parcurgerea iterativă a tabloului a şi, la fiecare parcurgere, ori de câte ori se întâlneşte o inversiune $(i, i+1)$ se procedează la interschimbarea $\mathtt{a[i]} \leftrightarrow \mathtt{a[i+1]}$. La prima parcurgere, elementul cel mai mare din secvenţă formează inversiuni cu toate elementele aflate după el şi, în urma interschimbărilor realizate, acesta va fi deplasat pe ultimul loc care este şi locul său final. La iteraţia următoare, la fel se va întâmpla cu cel de-al doilea element cel mai mare. În general, dacă subsecvenţa $\mathtt{a[r+1..n-1]}$ nu are nici o inversiune la iteraţia curentă, atunci ea nu va avea inversiuni la nici una din iteraţiile următoare. Aceasta permite ca la iteraţia următoare să fie verificată numai subsecvenţa $\mathtt{a[0..r]}$. Terminarea algoritmului este dată de faptul că la fiecare iteraţie numărul de interschimbari este micşorat cu cel puţin 1.

Descrierea algoritmului este următoarea:

```
procedure bubbleSort(a, n)
   ultim ← n-1
   while (ultim > 0) do
      n1 ← ultim - 1
      ultim ← 0
      for i ← 0 to n1 do
          if (a[i] > a[i+1])
             then swap(a[i], a[i+1])
                  ultim ← i
end
```

**Evaluarea algoritmului** Cazul cel mai favorabil este întâlnit atunci când secvenţa de intrare este deja sortată, caz în care algoritmul `bubbleSort` execută $O(n)$ operaţii. Cazul cel mai nefavorabil este obţinut când secvenţa de intrare este ordonată descrescător şi, în această situaţie, procedura execută $O(n^2)$ operaţii. În continuare, ne ocupăm de timpul mediu de execuţie. Mai precis, vom arăta că numărul mediu de comparaţii este

$$O(\frac{1}{2}(n^2 - n\log n - O(n)) + O(\sqrt{n}))$$

Fără a restrânge generalitatea, presupunem că valoarea iniţială a tabloului `a` este o permutare a mulţimii $\{1, 2, \ldots, n\}$ : $\langle v_0, \ldots v_{n-1}\rangle = Perm(0, 1, \ldots, n-1)$.

**Definiţia 4.1.** *Fie $x = \langle x_0, \ldots, x_{n-1}\rangle$ o permutare a mulţimii $\{0, 1, ..., n-1\}$. Numim* cod *al permutării $x$ secvenţa $cod(x) = (y_0, \ldots, y_{n-1})$, unde $y_j$ reprezintă numărul elementelor $x_i$ aflate la stânga lui $j$ şi mai mari decât $j$, i.e.*

$$y_j = \#\{x_i \mid i < k \text{ şi } x_i > j\},$$

*unde $k$ satisface $x_k = j$. Secvenţa $cod(x)$ mai este numită şi* tablou de inversări. *Vom prefera totuşi denumirea de cod al permutării.*

Codurile permutărilor ne vor ajuta la studiul proprietăţilor algoritmului `BubbleSort`. Să observăm mai întâi că valoarea $cod(v)(j)$ reprezintă numărul de inversiuni ale căror componentă secundară este $j$. De exemplu, dacă $v = (4, 2, 3, 1, 0)$, atunci $cod(v) = (4, 3, 1, 1, 0)$. După o trecere a algoritmului `BubbleSort` (o execuţie a buclei `while`) se obţine următoarea valoare pentru `a`: $w = (2, 3, 1, 0, 4)$. Avem $cod(w) = (3, 2, 0, 0, 0)$. Se observă următoarea proprietate:

$$cod(v)(i) \neq 0 \Rightarrow cod(w)(i) = cod(v)(i) - 1$$

În general are loc:

**Teorema 4.1.** *Presupunem că valoarea tabloului* `a` *este $v = (v_0, \ldots, v_{n-1}) = Perm(0, 1, \ldots, n-1)$ şi că, după o trecere a algoritmului* `BubbleSort`, *se obţine valoarea $w = (w_0, \ldots, w_{n-1})$ pentru* `a`. *Atunci are loc:*

$$(\forall i) cod(v)(i) \neq 0 \Rightarrow cod(w)(i) = cod(v)(i) - 1$$

Teorema de mai sus justifică atât proprietatea de terminare a algoritmului `BubbleSort`, cât şi corecta utilizare a variabilei `ultim`. În continuare ne ocupăm de calculul numărului mediu de comparaţii. Plecăm de la presupunerea că orice permutare $v$ a mulţimii $\{0, 1, ..., n-1\}$ poate să apară cu aceeaşi probabilitate, $p = \frac{1}{n!}$, ca intrare a algoritmului `BubbleSort`. Notăm cu $u_j$ valoarea expresiei `ultim-1` (variabilei `n1` la începutul iteraţiei $j$). Fie $w$ valoarea tabloului `a` obţinută după cea de-a $(j-1)$-a iteraţie. Dacă $cod(w)(i) > 0$, atunci există $cod(w)(i)$ elemente mai mari decât $i$ aflate la stânga lui $i$; cel mai mare dintre aceste elemente va avea poziţia finală $p \geq cod(w)(i) + i$. Mai mult, dacă elementul $i$ este cel mai din dreapta care trebuie schimbat la iteraţia $j$, atunci are loc:

$$cod(w)(i) \geq 1 \text{ şi } ultim = cod(w)(i) + i$$

De aici rezultă $u_j = \max_i\{cod(w)(i) + i - 1 \mid cod(w)(i) \geq 1\}$. Aplicând teorema de mai sus de $j-1$ ori, obţinem: $cod(w)(i) = cod(v)(i) - (j-1)$. Relaţia de mai sus devine $u_j = \max_i\{cod(v)(i) + i - j \mid cod(v)(i) > j-1\}$. Pentru un $k$ dat, algoritmul `bubbleSort` va executa cel mult $k$ iteraţii pentru acele intrări care satisfac $\forall i \; cod(v)(i) < k$. Numărul de permutări care satisfac proprietatea de mai sus este egal cu $k^{n-k}k!$ (conform [Knu76] p. 108). Rezultă că probabilitatea ca algoritmul `BubbleSort` să execute cel mult $k$ iteraţii este $\frac{1}{n!}(k^{n-k}k!)$, iar probabilitatea ca să se execute exact $k$ iteraţii este $\frac{1}{n!}(k^{n-k}k! - (k-1)^{n-k+1}(k-1)!)$. Pentru calcularea numărului mediu de comparaţii, vom determina valorile medii pentru $u_j$. Pentru aceasta, este util să determinăm numărul permutărilor $v$ cu proprietatea

$$(\forall i) cod(v)(i) < j-1 \text{ sau } cod(v)(i) + i - j \leq k$$

(numărul intrărilor pentru care `BubbleSort` execută cel mult $j+k$ iteraţii). Conform cu [Knu76] p. 108, acest număr este:

$$f_j(k) = (j+k)!(j-1)^{n-j-k}, \text{ pentru } j \leq j+k \leq n$$

Valoarea medie a lui $u_j$ este:

$$u_j^{\text{med}} = \frac{\sum_{(k|j<k+j\leq n)} k(f_j(k) - f_j(k-1))}{n!}$$

Conform cu [Knu76] p. 108 şi pp. 129-134 are loc:

$$C^{\text{med}}(n) = O(\frac{1}{2}(n^2 - n\log n - O(n)) + O(\sqrt{n}))$$

În cele ce urmează, vom prezenta strategia cunoscută sub numele de *par-impar* (*odd-even-sort*).Esenţa metodei *par-impar* constă în alternarea secvenţei de operaţii `swap(a[i],a[i+1]), i=0,2,4,...` cu secvenţa `swap(a[i],a[i+1]), i=1,3,5,...` După cel mult $n/2$ execuţii a fiecăreia din cele două secvenţe de operaţii, elementele tabloului `a` vor fi sortate. Metoda nu conduce la un algoritm superior algoritmului `bubbleSort`. Totuşi, algoritmul de sortare prin metoda *par-impar* are calitatea de a fi paralelizabil. Aceasta se datorează faptului că perechile care interacţionează, atunci când se execută una din cele două secvenţe de operaţii menţionate anterior, sunt disjuncte.

```
procedure par-imparSort(a, n)
   for s ← 0 to n-1 do
      if s este par
         then for i ← 0 to n-2 step 2 do
                  swap(a[i], a[i+1])
      if s este impar
         for i ← 1 to n-2 step 2 do
            swap(a[i], a[i+1])
end
```

## 4.1.2 Sortare prin inserţie

Una dintre familiile importante de tehnici de sortare se bazează pe metoda „jucătorului de bridge" (atunci când îşi aranjează cărţile), prin care fiecare element este inserat în locul corespunzător în raport cu elementele sortate anterior.

### 4.1.2.1 Sortare prin inserţie directă

Principiul de bază al algoritmului de sortare prin inserţie este următorul: Se presupune că subsecvenţa $(a[0], \ldots, a[j-1])$ este sortată. Se caută în această subsecvenţă locul $i$ al elementului $a[j]$ şi se inserează $a[j]$ pe poziţia $i$. Poziţia $i$ este determinată astfel:

- $i = 0$, dacă $a[j] < a[0]$;
- $0 < i < j$ şi satisface $a[i-1] \leq a[j] < a[i]$;
- $i = j$, dacă $a[j] \geq a[j-1]$.

Determinarea lui $i$ se poate face prin căutare secvenţială sau prin căutare binară. Considerăm cazul când poziţia $i$ este determinată prin căutare secvenţială (de la dreapta la stânga) simultan cu deplasarea elementelor mai mari decât $a[j]$ cu o poziţie la dreapta. Această deplasare se realizează prin interschimbări, astfel încât valoarea $a[j]$ realizează câte o deplasare la stânga până ajunge la locul ei final.

```
procedure insertSort(a, n)
   for j ← 1 to n-1 do
       i ← j-1
       temp ← a[j]
       while ((i ≥ 0) and (temp < a[i])) do
          a[i+1] ← a[i]
          i ← i-1
       if (i ≠ j-1) then a[i+1] ← temp
end
```

**Evaluarea.** Căutarea poziţiei $i$ în subsecvenţa a[0..j − 1] necesită $O(j-1)$ timp. Rezultă că timpul total în cazul cel mai nefavorabil este $O(1 + \cdots + n-1) = O(n^2)$. Pentru cazul cel mai favorabil, când valoarea tabloului la intrare este deja în ordine crescătoare, timpul de execuţie este $O(n)$.

**Exerciţiul 4.1.1.** Timpul de execuţie al algoritmului de sortare prin inserţie poate fi îmbunătăţit considerând secvenţa de sortat reprezentată printr-o listă simplu înlănţuită. Să se rescrie algoritmul `InsertSort` corespunzător acestei reprezentări. Să se precizeze timpul de execuţie al noului algoritm.

**Exerciţiul 4.1.2.** Să se rescrie algoritmul `InsertSort`, astfel încât căutarea poziţiei $i$ în subsecvenţa a[0..j − 1] să se facă prin metoda căutării binare. Să se analizeze timpul de execuţie al noului algoritm.

#### 4.1.2.2 Metoda lui Shell

În algoritmul precedent elementele se deplasează numai cu câte o poziţie o dată şi prin urmare timpul mediu va fi proporţional cu $n^2$, deoarece fiecare element călătoreşte în medie $n/3$ poziţii în timpul procesului de sortare. Din acest motiv s-au căutat metode care să îmbunătăţească inserţia directă, prin mecanisme cu ajutorul cărora elementele fac salturi mai lungi în loc de paşi mici. O asemenea metodă a fost propusă în anul 1959 de Donald L. Shell, metodă pe care o vom mai numi *sortare cu micşorarea incrementului*. Următorul exemplu ilustrează ideea generală care stă la baza metodei.

**Exemplu.** Presupunem $n = 9$. Sunt executaţi următorii paşi (figura 4.1):

1. *Prima trecere.* Se împart elementele în grupe de câte trei sau două elemente (valoarea incrementului $h_1 = 4$) care se sortează separat:

$$(\mathtt{a[0]}, \mathtt{a[4]}, \mathtt{a[8]}), \ldots, (\mathtt{a[3]}, \mathtt{a[7]})$$

2. *A doua trecere.* Se grupează elementele în două grupe de câte trei elemente (valoarea incrementului $h_2 = 3$): $(\mathtt{a[0]}, \mathtt{a[3]}, \mathtt{a[6]}) \ldots (\mathtt{a[2]}, \mathtt{a[5]}, \mathtt{a[8]})$ şi se sortează separat.

3. *A treia trecere.* Acest pas termină sortarea prin considerarea unei singure grupe care conţine toate elementele. În final, cele nouă elemente sunt sortate.

sfex

Fiecare din procesele intermediare de sortare implică fie o sublistă nesortată de dimensiune relativ scurtă, fie una aproape sortată, astfel că inserţia directă poate fi utilizată cu succes pentru fiecare operaţie de sortare. Prin aceste inserţii intermediare, elementele tind să conveargă rapid spre destinaţia lor finală. Secvenţa de incremente 4, 3, 1 nu este obligatorie; poate fi utilizată orice secvenţă $h_i > h_{i-1} > \cdots > h_0$, cu condiţia ca ultimul increment $h_0$ să fie 1.

Presupunem că numărul de valori de incrementare este memorat de variabila `nincr` şi că acestea sunt memorate în tabloul $(\mathtt{kval[h]} \mid 0 \leq h \leq nincr - 1)$. Subprogramul care descrie metoda lui Shell este:

```
procedure ShellSort(a, n)
begin
   for h ← nincr-1 downto 0 do
       k ← kval[h]
       for i ← k to n-1 do
           temp ← a[i]
           j ← i-k
           while ((j ≥ 0) and (temp < a[j])) do
              a[j + k] ← a[j]
              j ← j - k
           if (j+k ≠ i) then a[j+k] ← temp
end
```

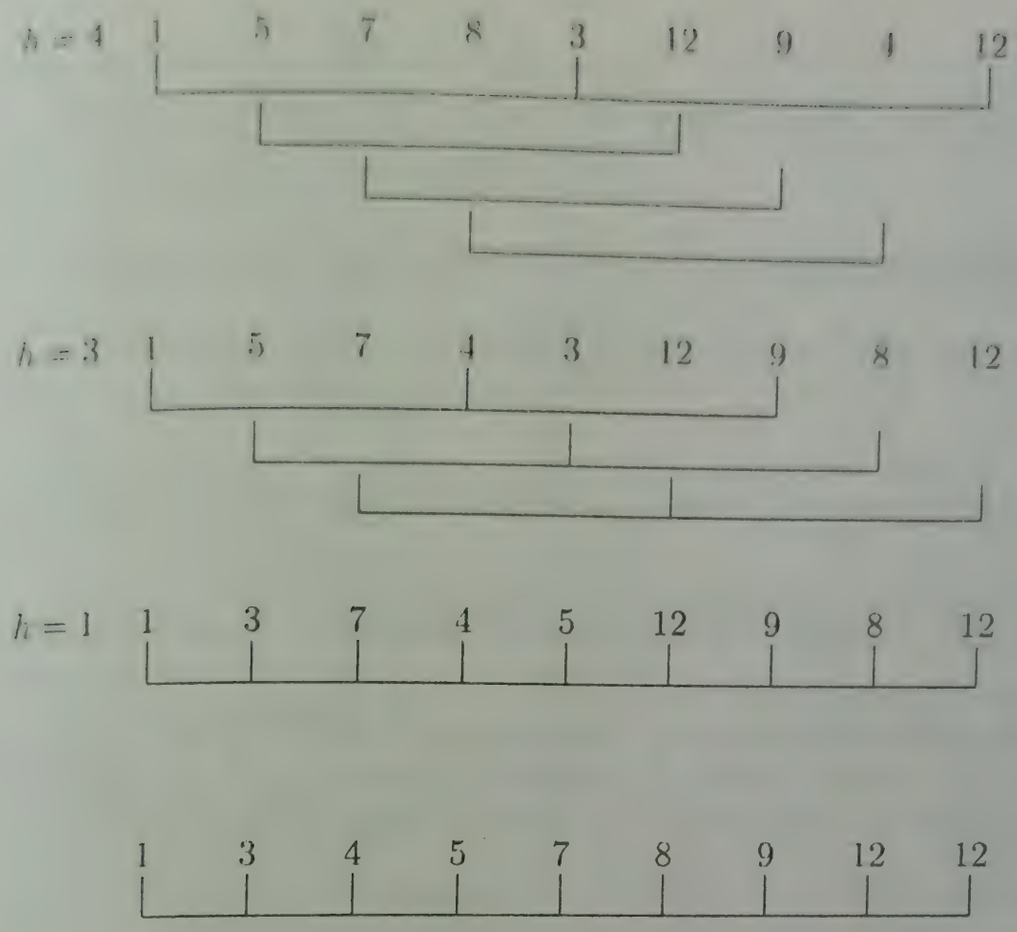



Figura 4.1: Sortarea prin metoda lui Shell

**Evaluarea metodei lui Shell.** Pentru evaluare vom presupune că elementele din secvenţă sunt diferite şi dispuse aleator. Vom denumi operaţia de sortare corespunzătoare primei treceri $h_t$-*sortare*, apoi $h_{t-1}$-*sortare* etc. O subsecvenţă pentru care $a[i] \leq a[i+h]$, pentru $0 \leq i \leq n-1-h$, va fi denumită *h-ordonată*. Vom considera pentru început cea mai simplă generalizare a inserţiei directe, şi anume cazul când avem numai două valori de incrementare: $h_1 = 2$ şi $h_0 = 1$. Cazul cel mai favorabil este obţinut când secvenţa de intrare este ordonată crescător şi sunt executate $\frac{n}{2} - 1 + n - 1$ comparaţii şi nici o deplasare. Cazul cel mai nefavorabil, când secvenţa de intrare este ordonată descrescător, necesită $\frac{1}{4}n(n-2) + \frac{n}{2}$ comparaţii şi tot atâtea deplasări (s-a presupus $n$ număr par). În continuare ne ocupăm de comportarea în medie. În cea de-a doua trecere avem o secvenţă 2-ordonată de elemente $a[0], a[1], \ldots, a[n-1]$. Este uşor de văzut că numărul de permutări $(i_0, i_1, \ldots i_{n-1})$ ale mulţimii $\{0, 1, \ldots, n-1\}$ cu proprietatea $i_k \leq i_{k+2}$, pentru $0 \leq k \leq n-3$, este $C_n^{\lfloor \frac{n}{2} \rfloor}$, deoarece obţinem exact o permutare 2-ordonată pentru fiecare alegere de $\lfloor \frac{n}{2} \rfloor$ elemente care să fie puse pe poziţii impare $1, 3, \ldots$. Fiecare permutare 2-ordonată este egal posibilă după ce o subsecvenţă aleatoare a fost 2-ordonată. Determinăm numărul mediu de inversări între astfel de permutări. Fie $A_n$ numărul total de inversări peste toate permutările 2-ordonate de $\{0, 1, \ldots, n-1\}$. Relaţiile $A_1 = 0$, $A_2 = 1$, $A_3 = 2$ sunt evidente. Considerând cele şase cazuri 2-ordonate

0 1 2 3 0 2 1 3 0 1 3 2 1 0 2 3 1 0 3 2 2 0 3 1

vom găsi $A_4 = 0 + 1 + 1 + 2 + 3 = 8$. În urma calculelor, care sunt un pic dificile (a se vedea [Knu76], p. 87), se obţine pentru $A_n$ o formă destul de simplă:

$$A_n = \left[\frac{n}{2}\right] 2^{n-2}$$

De aceea, numărul mediu de inversări într-o permutare aleatoare 2-ordonată este:

$$\frac{\left[\frac{n}{2}\right] 2^{n-2}}{C_n^{\lfloor \frac{n}{2} \rfloor}}$$

După aproximarea lui Stirling, aceasta converge asimptotic către $\frac{\sqrt{\pi}}{128 n^{\frac{3}{2}}} \approx 0.15 n^{\frac{3}{2}}$.

**Teorema 4.2.** *Numărul mediu de inversări executate de algoritmul lui Shell pentru secvenţa de incremente* $(2, 1)$ *este* $O(n^{\frac{3}{2}})$.

Se pune problema dacă în loc de secvenţa de valori de incrementare $(2, 1)$ se consideră $(h, 1)$, atunci pentru ce valori ale lui $h$ se obţine un timp cât mai mic pentru cazul cel mai nefavorabil. Are loc următorul rezultat ([Knu76], p. 89).

**Teorema 4.3.** *Dacă* $h \approx \left(\frac{16n}{\pi}\right)^{\frac{1}{3}}$, *atunci algoritmul lui Shell necesită timpul* $O(n^{\frac{5}{3}})$ *pentru cazul cel mai nefavorabil.*

Pentru cazul general, când secvenţa de valori de incrementare pentru algoritmul lui Shell este $h_{t-1}, \ldots, h_0$, se cunosc următoarele rezultate.

**Teorema 4.4.** *Dacă secvenţa de incremente* $h_{t-1}, \ldots, h_0$ *satisface condiţia*

$$h_{s+1} \bmod h_s = 0 \text{ pentru } 0 \leq s < t-1,$$

*atunci timpul de execuţie pentru cazul cel mai nefavorabil este* $O(n^2)$.

O justificare intuitivă a teoremei de mai sus este următoarea. De exemplu, dacă $h_s = 2^s$, $0 \leq s \leq 3$, atunci o 8-sortare urmată de o 4-sortare, urmată de o 2-sortare nu permite nici o interacţiune între elementele de pe poziţiile pare şi impare. De aceea, trecerii finale de 1-sortare îi vor reveni $O(n^{\frac{3}{2}})$ inversări. Dar să observăm că o 7-sortare urmată de o 5-sortare, urmată de o 3-sortare amestecă astfel lucrurile încât trecerea finală de 1-sortare nu va găsi mai mult de $2n$ inversări. Astfel, are loc următoarea teoremă.

**Teorema 4.5.** *Timpul de execuţie în cazul cel mai nefavorabil a algoritmului* `ShellSort` *este* $O(n^{\frac{3}{2}})$, *atunci când* $h_s = 2^{s+1} - 1$, $0 \leq s \leq t-1, t = [\log_2 n] - 1$.

### 4.1.3 Sortarea prin selecţie

Strategiile de sortare incluse în această clasă se bazează pe următoarea schemă : la pasul curent se selectează un element din secvenţă şi se plasează pe locul său final. Procedeul continuă până când toate elementele sunt plasate pe locurile lor finale. După modul în care se face selectarea elementului curent, metoda poate fi mai mult sau mai puţin eficientă. Noi ne vom ocupa doar de două strategii de sortare prin selecţie.

#### 4.1.3.1 Selecţia naivă

Este o metodă mai puţin eficientă, dar foarte simplă în prezentare. Se bazează pe următoarea caracterizare a predicatului $SORT(\mathtt{a})$:

$$SORT(\mathtt{a}) \iff (\forall i)(0 \leq i < n) \Rightarrow a[i] = \max\{a[0], \ldots, a[i]\}$$

Ordinea în care sunt aşezate elementele pe poziţiile lor finale este $n-1, n-2, \ldots, 0$. O formulare ehivalentă este:

$$SORT(\mathtt{a}) \iff (\forall i)(0 \leq i < n) : a[i] = \min\{a[i], \ldots, a[n]\},$$

caz în care ordinea de aşezare este $0, 1, \ldots, n-1$.

Subprogramul `naivSort` determină de fiecare dată locul valorii maxime:

```
procedure naivSort(a, n)
for i ← n-1 downto 1 do
    locmax ← 0
    maxtemp ← a[0]
    for j ← 1 to i do
        if (a[j] > maxtemp)
           then locmax ← j
                maxtemp ← a[j]
    a[locmax] ← a[i]
    a[i] ← maxtemp
end
```

**Evaluarea** algoritmului descris de procedura `naivSort` este simplă şi conduce la un timp de execuţie $O(n^2)$ pentru toate cazurile, adică algoritmul `NaivSort` are timpul de execuţie $\Theta(n^2)$. Este interesant de comparat `BubbleSort` cu `NaivSort`. Cu toate că sortarea prin metoda bulelor face mai puţine comparaţii decât selecţia naivă, ea este aproape de două ori mai lentă decât selecţia naivă, din cauza faptului că realizează multe schimbări în timp ce selecţia naivă implică o mişcare redusă a datelor. În tabelul din figura 4.2 sunt redaţi timpii de execuţie (în sutimi de secunde) pentru cele două metode obţinuţi în urma a zece teste pentru $n = 1000$.

#### 4.1.3.2 Selecţia sistematică

Se bazează pe structura de date de tip max-*heap* 2.8.2. Metoda de sortare prin selecţie sistematică constă în parcurgerea a două etape:

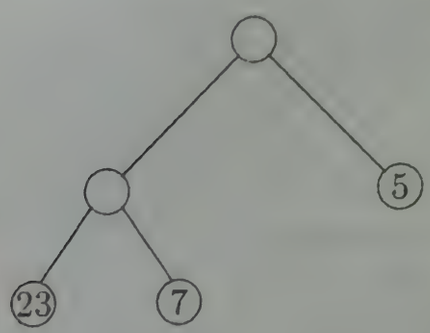
1. construirea pentru secvenţa curentă a proprietăţii MAX-HEAP(a);
2. selectarea în mod repetat a elementului maximal din secvenţa curentă şi refacerea proprietăţii MAX-HEAP pentru secvenţa rămasă.

*Etapa* 1. Considerăm că tabloul a are lungimea $n$. Iniţial, are loc MAX-HEAP(a, $\frac{n}{2}$).

**Exemplu.** Presupunem că valoarea tabloului a este $a = (10, 17, 5, 23, 7)$. Se observă imediat că are loc MAX-HEAP(a, 2):

| Nr. test | BubbleSort | NaivSort |
|---|---|---|
| 1 | 71 | 33 |
| 2 | 77 | 27 |
| 3 | 77 | 28 |
| 4 | 94 | 38 |
| 5 | 82 | 27 |
| 6 | 77 | 28 |
| 7 | 83 | 32 |
| 8 | 71 | 33 |
| 9 | 71 | 39 |
| 10 | 72 | 33 |

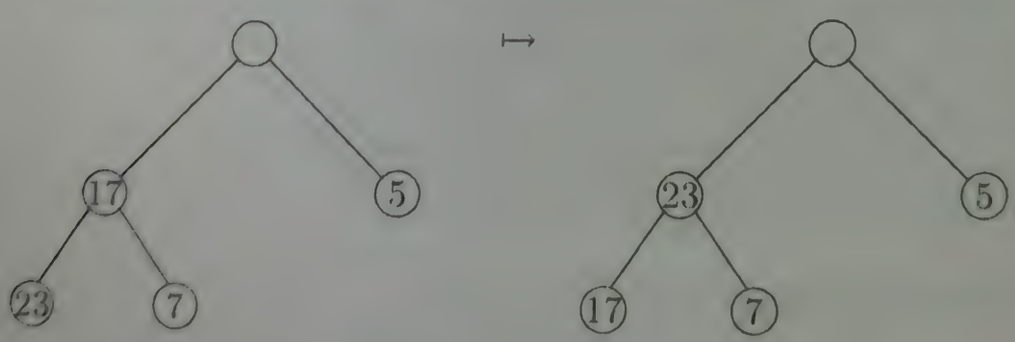
Figura 4.2: Compararea algoritmilor BubbleSort şi NaivSort




sfex

Dacă are loc MAX-HEAP(a, $\ell+1$) atunci se procedează la introducerea lui $a[\ell]$ în grămada deja construită a[$\ell+1$..n$-1$], astfel încât să obţinem MAX-HEAP(a, $\ell$). Procesul se repetă până când $\ell$ devine 0.

**Exemplu.** (Continuare) Avem $\ell = 1$ şi introducem pe $a[1] = 17$ în grămada a[2..4]:




Se obţine secvenţa $(10, 23, 5, 17, 7)$ care are proprietatea MAX-HEAP începând cu 1. Considerăm $\ell = 0$ şi introducem $a[0] = 10$ în grămada a[1..4]. Se obţine valoarea $(23, 17, 5, 10, 7)$ pentru tabloul a, valoare care verifică proprietatea MAX-HEAP.

sfex

Algoritmul de introducere a elementului $a[\ell]$ în grămada a[$\ell$ + 1..n $-$ 1], pentru a obţine MAX-HEAP($a, \ell$), este asemănător celui de inserare într-un max-*heap*:

```
procedure intrInGr(a, n, ℓ)
    j ← ℓ
```

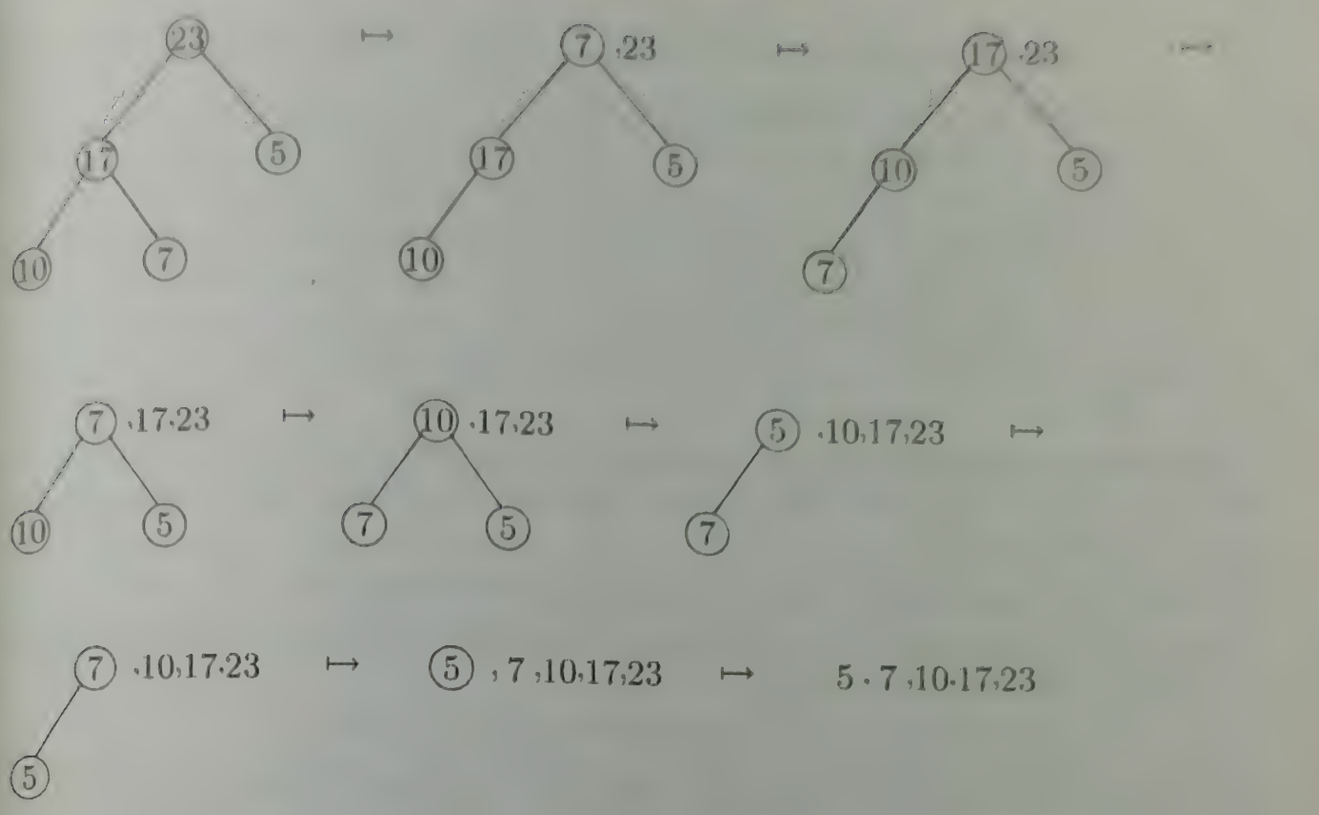



Figura 4.3: Etapa a doua

```
    esteHeap ← false
    while ((2*j+1 ≤ n-1) and not esteHeap)
       k ← j*2+1
       if ((k < n-1) and (a[k] < a[k+1])
          then k ← k+1
       if (a[j] < a[k])
          then swap(a[j], a[k])
          else esteHeap ← true
       j ← k
end
```

*Etapa* 2. Deoarece iniţial avem MAX-HEAP(a), rezultă că pe primul loc se găseşte elementul maximal din a[0..n − 1]. Punem acest element la locul său final prin interschimbarea a[0] ↔ a[n − 1]. Acum a[0..n − 2] are proprietatea MAX-HEAP începând cu 1. Refacem MAX-HEAP pentru această secvenţă prin introducerea lui $a[0]$ în grămada a[0..n − 2], după care punem pe locul său final cel de-al doilea element cel mai mare din secvenţa de sortat. Procedeul continuă până când toate elementele ajung pe locurile lor finale.

**Exemplu.** Etapa a doua pentru exemplul anterior este ilustrată în figura 4.3. sfex

Algoritmul de sortare prin selecţie sistematică este descris de subprogramul `heapSort`:

```
procedure heapSort(a,n)
```

```
    n1 ← ⌊(n-1)/2⌋
    for ℓ ← n1 downto 0 do
        intrInGr(a, n, ℓ)
    r ← n-1
    while (r ≥ 1) do
        swap(a[0], a[r])
        intrInGr(a, r, 0)
        r ← r-1
end
```

**Evaluarea algoritmului heapSort.** Considerăm $n = 2^k - 1$. În faza de construire a proprietății MAX-HEAP pentru toată secvența de intrare sunt efectuate următoarele operații:

- se construiesc vârfurile de pe nivelele $k-2, k-3, \ldots$;
- pentru construirea unui vârf de pe nivelul $i$ se vizitează cel mult câte un vârf de pe nivelele $i+1, \ldots, k-1$;
- la vizitarea unui vârf sunt executate două comparații.

Rezultă că numărul de comparații executate în prima etapă este cel mult:

$$\sum_{i=0}^{k-2} 2(k-i-1)2^i = (k-1)2 + (k-2)2^2 + \cdots + 1 \cdot 2^{k-1} = 2^{k+1} - 2(k+1)$$

În etapa a II-a, dacă presupunem că $a[r]$ se găseşte pe nivelul $i$, introducerea lui $a[0]$ în grămada a[1..r] necesită cel mult $2i$ comparații. Deoarece $r$ ia valori de la 1 la $n-1$, rezultă că în această etapă numărul total de comparatii este cel mult:

$$\sum_{i=0}^{k-1} 2i2^i = (k-2)2^{k+1} + 4$$

Numărul total de comparații este cel mult:

$$\begin{aligned} C(n) &= 2^{k+1} - 2(k+1) + (k-2)2^{k+1} + 4 \\ &= 2^{k+1}(k-1) - 2(k-1) \\ &= 2k(2^k - 1) - 2(2^k - 1) \\ &= 2n\log_2 n - 2n \end{aligned}$$

De unde rezultă că numărul de comparații este $C(n) = O(n \log_2 n)$.

### 4.1.4 Sortarea rapidă

A se vedea secțiunea 10.3.

### 4.1.5 Sortarea prin interclasare

A se vedea secțiunea 10.2.

### 4.1.6 Numărul minim de comparaţii pentru sortare

Algoritmii de sortare prezentaţi până acum se bazează pe executarea a două operaţii primitive: compararea şi interschimbarea a două elemente. Deoarece orice interschimbare este, în general, precedată de o comparaţie (prin care se decide dacă interschimbarea este necesară) putem spune că operaţiile de comparare domină calculul oricărui algoritm prezentat până acum. Ne punem următoarele două întrebări:

- care este numărul minim de comparaţii executate în cazul cel mai nefavorabil?
- care algoritmi de sortare realizează minimul de comparaţii, i.e. care algoritmi sunt optimali?

Pentru a putea răspunde la cele două întrebări trebuie mai întâi să precizăm modelul de calcul peste care sunt construiţi aceşti algoritmi.

Să analizăm câţiva algoritmi pentru ordonarea secvenţei $(a_0, a_1, a_2)$. Vom încerca să determinăm care elemente se compară pentru obţinerea permutării dorite. O primă observaţie pe care o facem este aceea că dacă se compară elementele $a_i$ şi $a_j$, notat $i\,?\,j$, atunci mulţimea comparaţiilor care urmează a fi făcute depinde de răspunsul obţinut prin efectuarea acestei comparaţii. Pentru simplitate vom presupune $a_i \neq a_j$ dacă $i \neq j$. Deoarece răspunsul dat de comparaţia $i\,?\,j$ are numai două posibilităţi de alegere, rezultă că putem reprezenta cele două mulţimi de comparaţii prin intermediul unui arbore binar:

- rădăcina va conţine o comparaţie $i\,?\,j$;
- subarborele din stânga conţine comparaţiile făcute în cazul $a_i < a_j$;
- subarborele din dreapta conţine comparaţiile făcute în cazul $a_i > a_j$.

Arborii corespunzători sortării prin selecţie naivă şi sortării prin inserţie sunt reprezentaţi în figura 4.4.

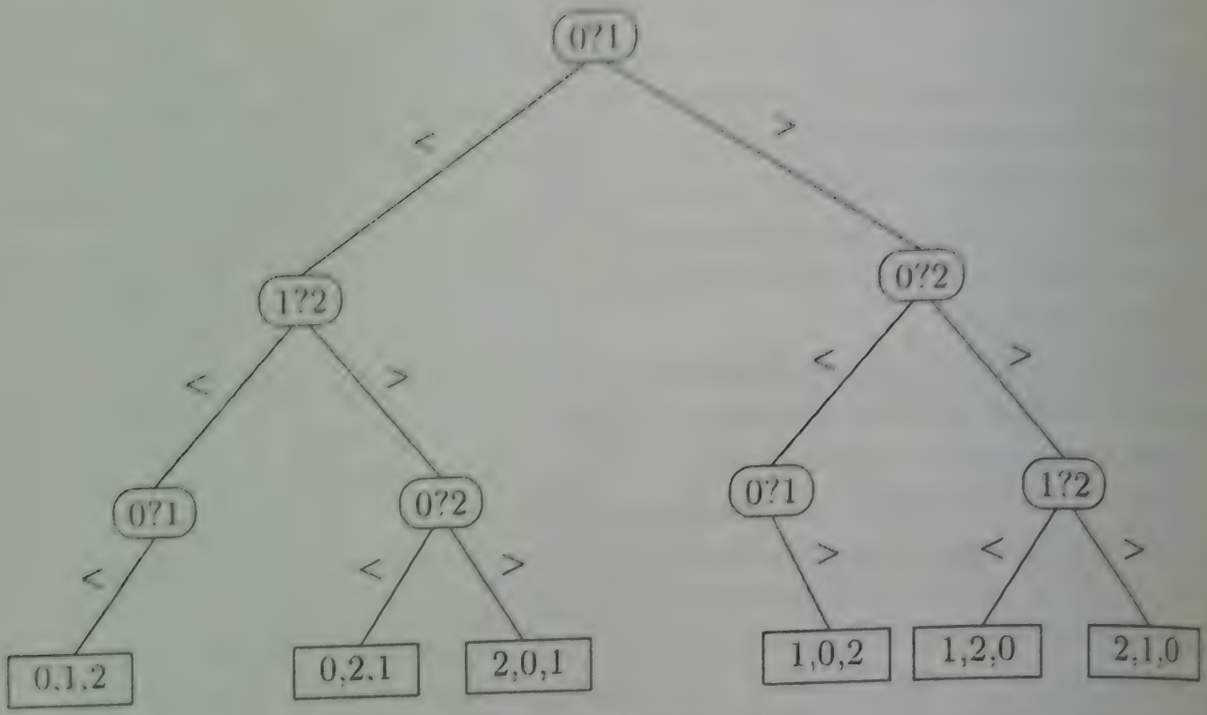
În vârfurile de pe frontieră am trecut permutările care dau ordinea crescătoare a elementelor. În acest fel putem asocia fiecărui algoritm de sortare bazat pe comparaţii un arbore binar. Vom utiliza aceşti arbori pentru a defini modelul de calcul.

**Definiţia 4.2.** *Un* arbore de decizie *pentru $n$ elemente este un arbore binar în care vârfurile interne sunt etichetate cu perechi de forma $i\,?\,j$, iar vârfurile de pe frontieră sunt etichetate cu permutări ale mulţimii $\{0, 1, \ldots, n-1\}$.*

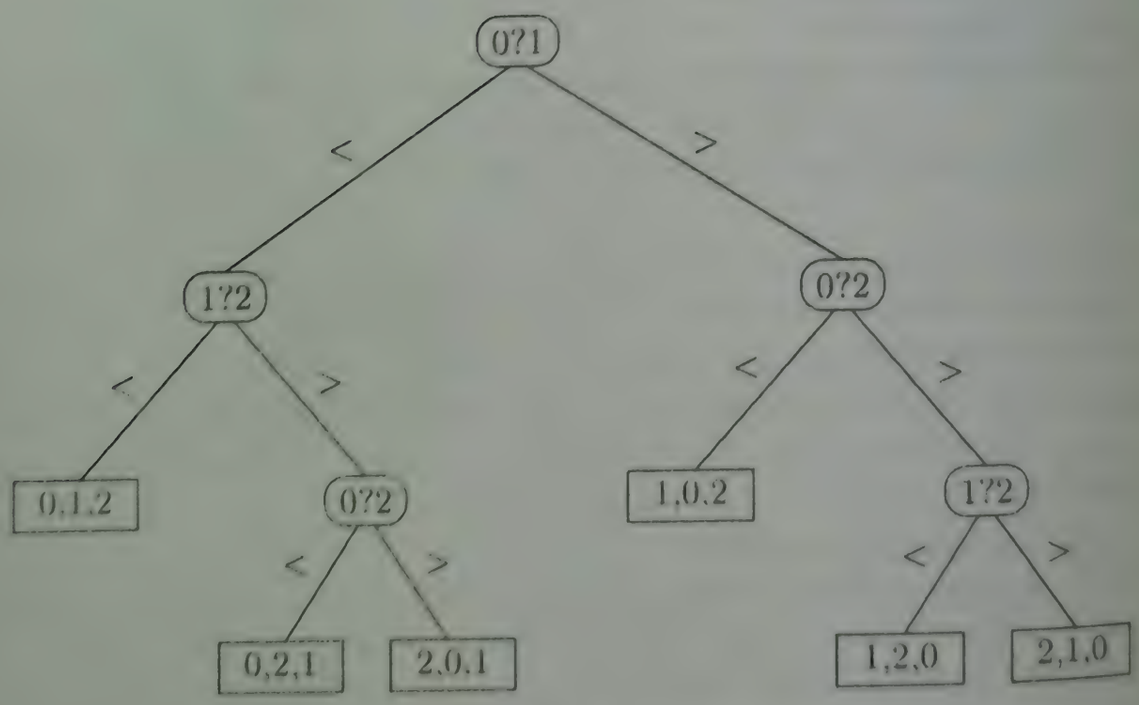
**Definiţia 4.3.** *Fie $t$ un arbore de decizie pentru $n$ elemente şi secvenţa $a = (a_0, \ldots, a_{n-1})$.* Calculul *lui $t$ pentru intrarea $a$ constă în parcurgerea unui drum de la rădăcină la un vârf de pe frontieră definit astfel:*

1. *Iniţial se pleacă din rădăcină.*
2. *Presupunem ca vârful curent este $i\,?\,j$. Dacă $a_i < a_j$, atunci fiul din stânga lui $i\,?\,j$ devine vârf curent; altfel, fiul din dreapta devine vârf curent.*
3. *Calculul se opreşte, dacă vârful curent este pe frontieră.*

**Definiţia 4.4.** *Fie $t$ un arbore de decizie pentru $n$ elemente. Spunem că $t$* rezolvă problema sortării, *dacă pentru orice intrare $a = (a_0, \ldots, a_{n-1})$, calculul lui $t$ pentru $a$ se termină într-un vârf etichetat cu permutarea $\pi$, care are proprietatea $a_{\pi(0)} < \ldots < a_{\pi(n-1)}$. Un arbore de decizie care rezolvă problema sortării va mai fi numit şi* arbore de decizie pentru sortare, *iar modelul de calcul va fi numit* modelul arborilor de decizie pentru sortare.




Sortare prin selecţie naivă




Sortare prin inserţie

Figura 4.4: Arbori de decizie pentru sortare

Putem defini acum timpul de execuţie minim pentru cazul cel mai nefavorabil prin expresia:

$$T(n) = \min_{t} \max_{\pi} \text{Lung}(\pi, t),$$

unde $\text{Lung}(\pi, t)$ reprezintă drumul de la rădăcină la vârful pe frontieră etichetat cu $\pi$ în arborele de decizie pentru sortare $t$.

**Teorema 4.6.** *Problema sortării are timpul de execuţie pentru cazul cel mai nefavorabil $\Omega(n \log n)$ în modelul arborilor de decizie pentru sortare.*

*Demonstraţie.* Un arbore de decizie pentru $n$ elemente, care rezolvă problema sortării, are $n!$ vârfuri pe frontieră. Un arbore de înălţime $k$ are cel mult $2^k$ vârfuri pe frontieră. De aici rezultă:

$$2^{T(n)} \geq n!$$

care implică $T(n) \geq \log_2(n!) = \Theta(n \log_2 n)$. sfdem

**Corolar 4.1.** *Algoritmul* `HeapSort` *este optimal în modelul arborilor de decizie pentru sortare.*

## 4.1.7 Exerciţii

**Exerciţiul 4.1.3.** Se consideră un tablou de structuri statice (`a[i]` | $0 \leq i < n$) şi un al doilea tablou (`b[i]` | $0 \leq i < n$) care conţine o permutare a mulţimii $\{0, 1, \ldots, n-1\}$. Să se scrie un program care rearanjează componentele tabloului `a` conform permutării date de `b`.

**Exerciţiul 4.1.4.** Să se modifice `ShellSort` astfel încât să utilizeze întotdeauna secvenţa de incremente $(h_0, \ldots, h_{k-1})$ dată prin recurenţa: $h_0 = 1; h_{i+1} = 3h_i + 1$ şi $h_{k-1}$ cel mai mare increment de acest fel mai mic decât $\frac{n}{2}$. Să se încerce apoi găsirea de alte secvenţe de incremente care să producă algoritmi de sortare mai eficienţi.

**Exerciţiul 4.1.5.** Să se scrie o variantă recursivă a algoritmului de inserare într-un heap `intrInGr`. Care dintre cele două variante este mai eficientă?

**Exerciţiul 4.1.6.** Care este timpul de execuţie al algoritmului `heapSort` dacă secvenţa de intrare este ordonată crescător? Dar dacă este ordonată descrescător?

**Exerciţiul 4.1.7.** Să se proiecteze un algoritm care să determine cel de-al doilea cel mai mare element dintr-o listă. Algoritmul va executa $n + \lceil \log n \rceil - 2$ comparaţii. *Indicaţie.* Fie $(a_0, \ldots, a_{n-1})$ secvenţa de intrare. Cel mai mare element din listă se poate face prin $n-1$ comparaţii. Se va utiliza următoarea metodă pentru determinarea maximului:

- se determină $b_0 = \max(a_0, a_1), b_1 = \max(a_2, a_3), \ldots$;
- se determină $c_0 = \max(b_0, b_1), c_1 = \max(b_2, b_3), \ldots$ etc.

Metodei de mai sus i se poate ataşa un arbore binar complet de mărime $n$, fiecare vârf intern reprezentând o comparaţie, iar rădăcina corespunzând celui mai mare element. Pentru a determina cel de-al doilea cel mai mare element este suficient să se considere numai elementele care au fost comparate cu maximul. Numărul acestora este $\log_2 n - 1$. Va trebui proiectată o structură de date pentru memorarea acestor elemente.

**Exerciţiul 4.1.8.** (Selecţie) Să se generalizeze metoda din exerciţiul anterior pentru a determina cel de-al $k$-lea cel mai mare element dintr-o listă. Care este timpul de execuţie al algoritmului? (Se ştie că există algoritmi care rezolvă această problemă în timpul $\Theta(n + \min(k, n-k)\log n)$).

**Exerciţiul 4.1.9.** (Sortare prin metoda turneelor.) Să se utilizeze algoritmul din exerciţiul precedent pentru sortarea unei liste. Care este timpul de execuţie al algoritmului de sortare?

**Exerciţiul 4.1.10.** Să se proiecteze programarea unui turneu de tenis la care participă 16 jucători astfel încât numărul de meciuri să fie minim, iar primii doi să fie corect clasaţi relativ la relaţia de ordine „$a$ mai bun decât $b$", definită astfel:

- dacă $a$ îl învinge pe $b$, atunci $a$ este mai bun decât $b$;
- dacă $a$ este mai bun decât $b$ şi $b$ mai bun decât $c$, atunci $a$ e mai bun decât $c$

## 4.2 Sortare prin distribuire

În următoarele două secţiuni vom studia algoritmi de sortare pentru cazurile când se cunosc informaţii suplimentare despre elementele care se sortează. Aşa cum reiese şi din denumire, algoritmii de sortare prin distribuire presupun cunoaşterea de informaţii privind distribuţia acestor elemente. Aceste informaţii sunt utilizate pentru a distribui elementele secvenţei de sortat în „pachete" care vor fi sortate în acelaşi mod sau prin altă metodă, după care pachetele se combină pentru a obţine lista finală sortată.

### 4.2.1 Sortarea cuvintelor

Presupunem că avem $n$ fişe, iar fiecare fişă conţine un nume ce identifică în mod unic fişa (cheia). Se pune problema sortării manuale a fişelor. Pe baza experienţei câştigate se procedează astfel: se împart fişele în pachete, fiecare pachet conţinând fişele ale căror cheie începe cu aceeaşi literă. Apoi se sortează fiecare pachet în aceeaşi manieră după a doua literă, apoi etc. După sortarea tuturor pachetelor, acestea se combină astfel încât formează o listă liniară sortată.

Vom încerca să formalizăm metoda de mai sus într-un algoritm de sortare a şirurilor de caractere (cuvinte). Presupunem că elementele secvenţei de sortat sunt şiruri de lungime fixată $m$ definite peste un alfabet cu $k$ litere. Echivalent, se poate presupune că elementele de sortat sunt numere reprezentate în baza $k$. Din acest motiv, sortarea cuvintelor este denumită în engleză *radix-sort* (cuvântul *radix* traducându-se prin *bază*).

Dacă urmăm ideea din exemplul cu fişele, atunci algoritmul ar putea fi descris recursiv astfel:

1. Se împart cele $n$ cuvinte în $k$ pachete, cuvintele din acelaşi pachet având aceeaşi literă pe poziţia $i$ (numărând de la stânga la dreapta).
2. Apoi fiecare pachet este sortat în aceeaşi manieră după literele de pe poziţiile $i+1, \ldots, m-1$.

3. Se concatenează cele $k$ pachete în ordinea dată de literele de pe poziţia $i$. Lista obţinută este sortată după subcuvintele formate din literele de pe poziţiile $i, i+1, \ldots, m-1$.

Iniţial se consideră $i = 0$. Apare următoarea problemă. Un grup de $k$ pachete nu va putea fi combinat într-o listă sortată decât dacă cele $k$ pachete au fost sortate complet pentru subcuvintele corespunzătoare. Este deci necesară ţinerea unei evidenţe a pachetelor, fapt care conduce la utilizarea de memorie suplimentară şi creşterea gradului de complexitate a metodei. O simplificare majoră apare dacă împărţirea cuvintelor în pachete se face parcurgând literele acestora de la dreapta la stânga. Procedând aşa, observăm următorul fapt surprinzător: după ce cuvintele au fost distribuite în $k$ pachete după litera de pe poziţia $i$, cele $k$ pachete pot fi combinate înainte de a le distribui după litera de pe poziţia $i-1$.

**Exemplu.** Presupunem că alfabetul este $\{0 < 1 < 2\}$ şi $m = 3$. Cele trei faze care cuprind distribuirea elementelor listei în pachete şi apoi concatenarea acestora într-o singură listă sunt sugerate grafic în figura 4.5. sfex

**Observaţie.** Atunci când se face distribuirea cuvintelor în pachete, ordinea apariţiilor cuvintelor într-un pachet trebuie să coincidă cu ordinea din lista iniţială. Dacă distribuirea se face după cheia $i$ şi două chei au pe poziţia $i$ aceeaşi literă, atunci o cheie o va preceda pe cealaltă în pachet dacă şi numai dacă o precedă şi în secvenţa iniţială (cea supusă sortării). Această proprietate este un caz particular al celei cunoscute sub numele de *stabilitate a algoritmilor de sortare.* sfobs

Pentru gestionarea pachetelor vom utiliza un tablou de pointeri numit `pachet`, cu semnificaţia că elementul *pachet*$[i]$ face referire la primul element din lista ce reprezintă pachetul $i$. Subetapa de distribuire este realizată în modul următor: Iniţial, se consideră listele *pachet*$[i]$ vide. Apoi se parcurge secvenţial lista supusă distribuirii şi fiecare element al acesteia este distribuit în pachetul corespunzător. Subetapa de combinare a pachetelor constă în concatenarea celor $k$ liste *pachet*$[i]$, $i = 0, \ldots, k-1$. Pentru ca operaţia de inserare a unei noi chei într-un pachet să se realizeze uşor, este util să folosim încă un tablou de pointeri `ultim` cu semnificaţia că *ultim*$[i]$ face referire la ultimul element din lista corespunzătoare pachetului $i$. De asemenea, tabloul `ultim` va fi utilizat cu succes şi la concatenarea listelor. Descrierea procedurală a algoritmului este următoarea:

```
procedure radixSort(a,n)
begin
   for i ← m-1 downto 0 do
       for j ← 0 to k-1 do
           pachet[j] ← listaVida()
       while (not esteVida(L)) do
          w ← citeste(L, 0)
          elimina(L, 0)
          insereaza(pachet[w[i]],lung(pachet[w[i]]), w)
       for j ← 0 to k-1 do
           concateneaza(L, pachet[j])
end
```

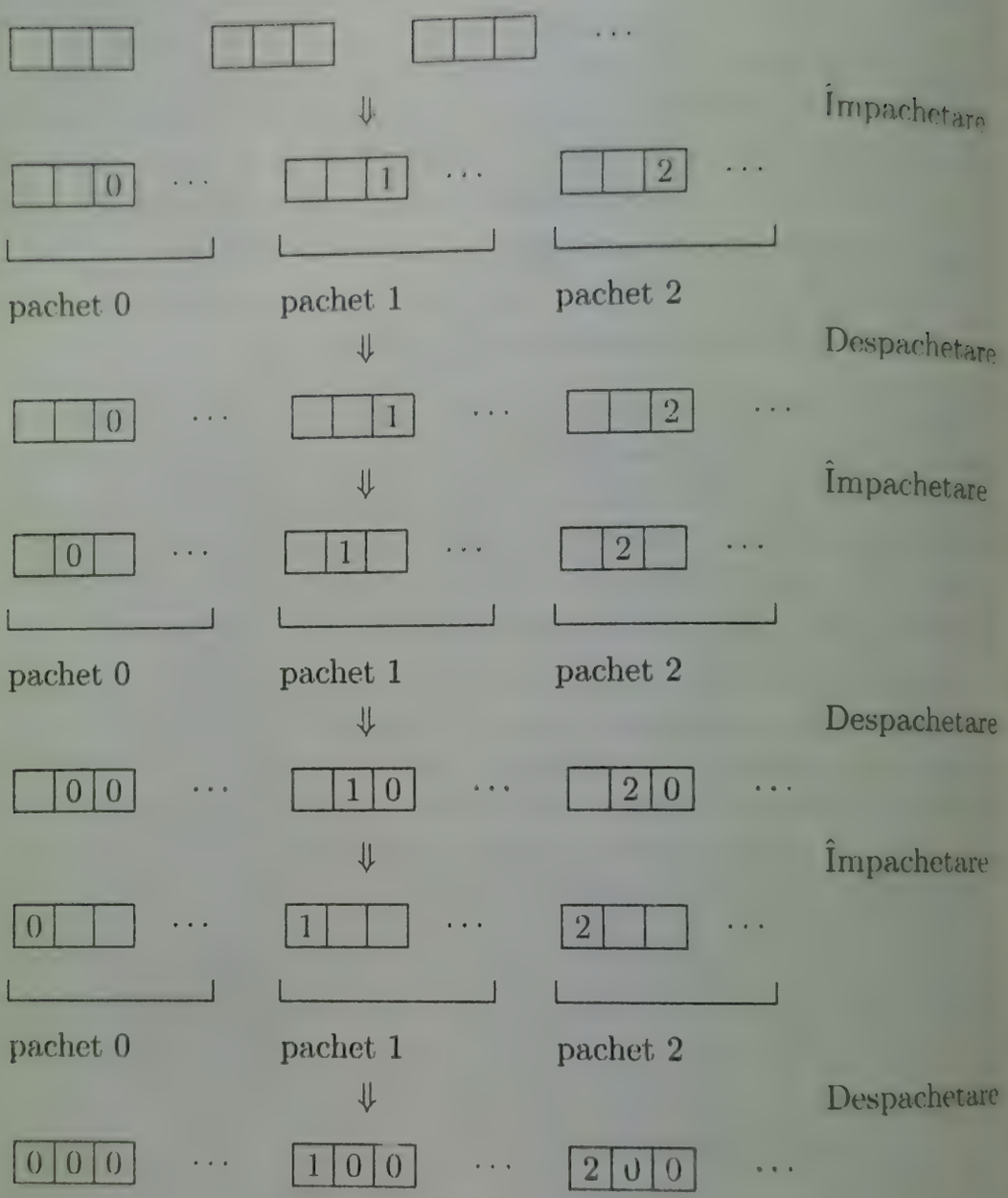



Figura 4.5: Sortare prin distribuire

Procedura `concateneaza(L1, L2)` concatenează listele L1 şi L2 şi memorează rezultatul în L1.

**Evaluarea algoritmului radixSort.** Distribuirea în pachete presupune parcurgerea completă a listei de intrare, iar procesarea fiecărui element al listei necesită $O(1)$ operaţii. Rezultă că faza de distribuire se face în timpul $O(n)$, unde $n$ este numărul de elemente din listă. Combinarea pachetelor presupune o parcurgere a tabloului `pachet`, iar adăugarea unui pachet se face cu $O(1)$ operaţii, cu ajutorul tabloului `ultim`. Astfel, faza de combinare a pachetelor necesită $O(k)$ timp. Rezultă că algoritmul `RadixSort` are un timp de execuţie de $O(m \cdot n)$. De remarcat că, atunci când $m$ are valori foarte mici comparativ cu $n$ ($O(\frac{n}{m}) = \Omega(n)$), se obţine o limită inferioară a valorii optime pentru modelul arborilor de decizie. Această îmbunătăţire este posibilă datorită informaţiei suplimentare despre organizarea cheilor supuse sortării.

## 4.2.2 Distribuire prin segmentare

Algoritmul de sortare care utilizează distribuirea prin segmentare (în engleză *bucket-sort*) foloseşte faptul că secvenţa de intrare este furnizată de un proces aleator care distribuie elementele uniform peste intervalul $[0, 1)$. Aşadar, presupunem că elementele de sortat sunt numere din acest interval. Ideea algoritmului este următoarea:

- Se divide intervalul $[0, 1)$ în $n$ subintervale de mărimi egale.
- Se distribuie cele $n$ intrări în $n$ pachete, un pachet conţinând numerele ce aparţin unui subinterval. Pachetul căruia îi aparţine intrarea $a_i$ este determinat de valoarea expresiei $\lceil n \cdot a_i \rceil + 1$, dacă pachetele sunt numerotate de la 1 la $n$. Deoarece intrările sunt distribuite uniform peste intervalul $[0, 1)$, este de aşteptat ca un pachet să nu conţină prea multe numere.
- Se sortează fiecare pachet utilizând o altă metodă, de exemplu, sortarea prin inserţie directă.
- Se combină cele $n$ pachete într-o listă sortată.

**Exerciţiul 4.2.1.** Presupunem că lista de intrare este reprezentată de un tablou. Să se scrie un subprogram `BucketSort` care descrie algoritmul de sortare de mai sus. Pachetele vor fi gestionate cu liste înlănţuite, ca în cazul algoritmului `RadixSort`. Sortarea unui pachet se va face prin inserţie directă.

Vom arăta că algoritmul `BucketSort` are timpul mediu de execuţie $O(n)$. Notăm cu $n_i$ variabila aleatoare care, pentru o intrare dată, dă numărul de elemente aparţinând intervalului $i$. Timpul de execuţie al algoritmului de sortare prin inserţie directă este $O(n^2)$ şi deci timpul mediu de execuţie pentru sortarea pachetului $i$ este $M[O(n_i^2)] = O(M[n_i^2])$. Rezultă că algoritmul `BucketSort` are timpul mediu de execuţie $\sum_{i=1}^{n} O(M[n_i^2]) = O(\sum_{i=1}^{n} M[n_i^2])$. Valorile posibile pentru variabila aleatoare $n_i$ sunt $0, 1, \ldots, n$, iar probabilitatea ca $n_i = k$ este $p_k = C_n^k (\frac{1}{n})^k (1 - \frac{1}{n})^{n-k}$ (distribuţie binomială). Rezultă că media variabilei aleatoare $n_i$ este $M[n_i] = \sum_{k=0}^{n} k \cdot p_k = n \cdot \frac{1}{n} = 1$. Pentru calculul mediei variabilei aleatoare $n_i^2$ vom utiliza formula $M[n_i^2] = D[n_i] + M^2[n_i]$, unde $D[n_i]$ este dispersia variabilei aleatoare $n_i$.

Se știe că dispersia unei variabile aleatoare cu distribuție binomială este $np(1-p)$, de unde $D[n_i] = n\frac{1}{n}(1-\frac{1}{n}) = 1 - \frac{1}{n}$. Rezultă $M[n_i^2] = 2 - \frac{1}{n} = \Theta(1)$, care implică $O(\sum_{i=1}^{n} M[n_i^2]) = O(\sum_{i=1}^{n} \Theta(1)) = O(n)$. De aici timpul mediu de execuție a algoritmului `BucketSort` este $O(n)$.

### 4.2.3 Exerciții

**Exercițiul 4.2.2.** Să se descrie un algoritm care sortează cuvinte, astfel încât distribuirea în pachete să se facă prin parcurgerea cuvintelor de la stânga la dreapta. Se presupune că lista cuvintelor este reprezentată în memorie printr-un tablou. Algoritmul va utiliza un singur tablou suplimentar de lungime $n$ și un singur tablou suplimentar de lungime $k$ și va avea timpul de execuție $O(m \cdot n)$.

**Exercițiul 4.2.3.** Să se aplice RadixSort la sortarea numerelor întregi, ținând cont de reprezentarea lor în memoria calculatorului.
*Indicație.* Reprezentarea binară a unui număr este divizată, de exemplu, în patru părți. Astfel un întreg poate fi văzut ca o cheie formată din patru „litere" (numere aparținând unui domeniu restrâns). Să se testeze eficiența noului algoritm de sortare cu cele ale algoritmilor bazați pe comparații.

**Exercițiul 4.2.4.** [CLR93] Să se utilizeze inducția pentru a dovedi că `RadixSort` rezolvă corect problema sortării. Unde este utilizată proprietatea că sortările intermediare sunt stabile?

**Exercițiul 4.2.5.** Să se scrie o variantă recursivă a algoritmului `RadixSort` în care ordonarea cuvintelor după literele de pe o anumită poziție se face prin interschimbare: se pleacă cu un indice $i$ din stânga și cu un indice $j$ din dreapta și se înaintează alternativ până când se întâlnește o inversiune. Se rezolvă inversiunea prin interschimbare după care procesul continuă până se parcurge toată secvența (a se vedea și pasul de divizare de la `QuickSort` din secțiunea 10.3). Mai există diferențe semnificative între cele două moduri de considerare a literelor după care se face ordonarea: de la stânga la dreapta și de la dreapta la stânga?

**Exercițiul 4.2.6.** [CLR93] Să se proiecteze un algoritm care să sorteze $n$ întregi din intervalul $[0, n^2)$ în timpul $O(n)$.
*Indicație.* Numerele din intervalul $[0, n^2)$ pot fi scrise ca secvențe de două cifre în baza $\log_2 n$ și se aplică algoritmul `RadixSort`.

**Exercițiul 4.2.7.** Să se arate că algoritmul `BucketSort` rezolvă corect problema sortării.

**Exercițiul 4.2.8.** Care este timpul de execuție al algoritmului `BucketSort` pentru cazul cel mai nefavorabil? Poate fi modificat `BucketSort` pentru a avea un timp de execuție de $O(n \log n)$ pentru cazul cel mai nefavorabil? (Modificarea va trebui să păstreze timpul mediu de execuție liniar).

## 4.3 Sortare prin numărare

Presupunem că tipul `TElement`, peste care sunt definite secvențele supuse sortării, conține numai două elemente: *alb* $<$ *negru*. Sortarea unui tablou `a : array[1..n]`

`oITElement` este foarte simplă: se determină numărul $n_1$ de elemente din `a` egale cu *alb* şi numărul $n_2$ de elemente egale cu *negru*. Aceasta se poate face în timpul $\Theta(n)$. Apoi se pun în tablou $n_1$ elemente *alb* urmate de $n_2$ elemente *negru*. Şi cea de-a doua etapă necesită $\Theta(n)$ timp. Rezultă un algoritm de sortare cu timpul de execuţie $\Theta(n)$. Şi de această dată, obţinerea unui algoritm de sortare liniar a fost posibilă datorită faptului că s-au cunoscut informaţii suplimentare despre elementele supuse sortării, anume că mulţimea univers conţine numai două elemente. Algoritmul de sortare prin numărare are ca punct de plecare observaţia de mai sus şi se utilizează atunci când mulţimea univers conţine puţine elemente. Prin simplificare, presupunem că $\mathbb{U} = \{0, 1, \ldots, k-1\}$ cu relaţia de ordine naturală. Algoritmul de la începutul secţiunii are un defect: dacă secvenţa de sortare este una de structuri statice şi sortarea se face după chei, atunci etapa a doua a algoritmului devine inefectivă. În plus, algoritmul nu are proprietatea de stabilitate. Vom modifica algoritmul astfel încât elementele tabloului de intrare `a` să fie transferate direct în tabloul de ieşire `b` pe locurile lor finale. Cum poate fi utilizată numărarea pentru a determina locurile elementelor în tabloul sortat? Revenim la exemplul de la începutul secţiunii. Observăm că elementul *negru* aflat cel mai la dreapta în `a` are poziţia $n_1 + n_2$ în `b`, următorul va avea poziţia $n_1 + n_2 - 1$ şi aşa mai departe. Elementul *alb* aflat cel mai la dreapta în `a` are poziţia $n_1$ în `b`, următorul poziţia $n_1 - 1$ şi aşa mai departe. Presupunem existenţa unui vector `p` cu valoarea iniţială $p[alb] = n_1, p[negru] = n_1 + n_2$. Etapa a doua a algoritmului va parcurge tabloul `a` de la dreapta la stânga şi va păstra invariantă următoarea proprietate: $p[alb]$ va fi poziţia primului element *alb* ce va fi întâlnit şi $p[negru]$ va fi poziţia primului element *negru* ce va fi întâlnit. Această proprietate poate fi păstrată prin decrementarea componentei din `p` imediat după ce elementul corespunzător din `a` a fost transferat. Prima etapă a algoritmului va determina vectorul `p` unde $p[i]$ = numărul de elemente din `a` mai mici decât sau egale cu $i$.

```
procedure sortNumarare(a, b, n)
begin
   for i ← 0 to k-1 do
       p[i] ← 0
   for j ← 0 to n-1 do
       p[a[j]] ← p[a[j]]+1
   for i ← 1 to k-1 do
       p[i] ← p[i-1] + p[i]
   for j ← n-1 downto 0 do
       i ← a[j]
       b[p[i]-1] ← i
       p[i] ← p[i]-1
end
```

Determinarea tabloului `p` necesită $O(k + n)$ timp, iar transferul $O(n)$ timp. Rezultă că algoritmul `sortNumarare` are timpul de execuţie $O(n + k)$. În practică, algoritmul este aplicat pentru $k = O(n)$, caz în care rezultă un timp de execuţie liniar a pentru `sortNumarare`.

### 4.3.1 Exerciţii

**Exerciţiul 4.3.1.** Să se arate că algoritmul `sortNumarare` este stabil.

**Exerciţiul 4.3.2.** Presupunem că în etapa a doua a algoritmului `sortNumarare` se înlocuieşte `for j ← n-1 downto 0` cu `for j ← 0 to n-1`. Să se arate că şi noul algoritm rezolvă corect problema sortării. Mai este noul algoritm stabil?

**Exerciţiul 4.3.3.** [CLR93] Să se proiecteze un algoritm care, pentru o secvenţă de $n$ întregi aparţinând intervalului $[0, k-1]$, preprocesează secvenţa în timpul $O(n+k)$, după care răspunde la întrebări de forma „câte elemente din secvenţă sunt într-un interval $[a, b]$" în timpul $O(1)$.

**Exerciţiul 4.3.4.** Presupunem că despre secvenţa de intrare se cunoaşte numai informaţia că elementele sunt distincte două câte două. Să se proiecteze un algoritm bazat pe numărare care să sorteze o astfel de secvenţă. Care este timpul de execuţie al algoritmului proiectat? Mai este posibilă proiectarea unui algoritm de sortare liniar utilizând numai această informaţie suplimentară despre datele de intrare?

## 4.4 Sortare topologică

Presupunem acum că relaţia de ordine este parţială. Fie, de exemplu, $a_1 < a_0, a_1 < a_2 < a_3$. Problema constă în a crea o listă liniară care să fie compatibilă cu relaţia de ordine: dacă $a_i < a_j$, atunci $a_i$ va precede pe $a_j$ în lista finală. Pentru exemplul nostru, lista liniară finală va putea fi $(a_1, a_0, a_2, a_3)$, sau $(a_1, a_2, a_0, a_3)$, sau $(a_1, a_2, a_3, a_0)$.

**Definiţia 4.5.** *Fie $(S, \leq)$ o mulţime parţial ordonată finită şi $a = (a_0, a_1, \ldots, a_{n-1})$ o liniarizare a sa. Spunem că secvenţa $a$ este* sortată topologic, *dacă $\forall i, j : a_i < a_j \Rightarrow i < j$.*

**Teorema 4.7.** *Orice mulţime parţial ordonată finită $(S, \leq)$ poate fi sortată topologic.*

*Demonstraţie.* Vom extinde relaţia $\leq$ la o relaţie de ordine totală. Fie elementele distincte $a, b \in S$ ce nu pot fi comparate, *i.e.*, nu are loc nici $a < b$ şi nici $b < a$. Extindem relaţia $<$, considerând $x < y$ pentru orice $x$ cu $x \leq a$ şi orice $y$ cu $b \leq y$. Procedeul continuă până când sunt eliminate toate perechile incomparabile. O mulţime total ordonată poate fi sortată, iar secvenţa sortată respectă ordinea parţială.

sfdem

Există o legătură strânsă între mulţimile parţial ordonate finite şi digrafurile aciclice (digrafuri fără circuite numite pe scurt dag-uri). Orice mulţime parţial ordonată $(S, \leq)$ defineşte un dag $D = (S, A)$, unde există arc de la $a$ la $b$, dacă $a < b$ şi nu există $c \in S$ cu proprietatea $a < c < b$. Reciproc, orice dag $D = (V, A)$ defineşte o relaţie de ordine parţială $\leq$ peste $V$, dată prin: $u \leq v$, dacă există un drum de lungime $\geq 0$ de la $u$ la $v$. De fapt, $\leq$ este închiderea reflexivă şi tranzitivă a lui $A$ (se mai notează $\leq = A^*$). Sortarea topologică a unui dag constă în crearea unei liste liniare a vârfurilor cu următoarea proprietate: dacă există arc de la $u$ la $v$, atunci $u$ precedă pe $v$ în listă, pentru oricare două vârfuri $u$ şi $v$. Vârfurile care

candidează pentru primul loc în lista sortată topologic au proprietatea că nu există arce incidente spre interior (care sosesc în acel vârf) şi se numesc *surse*.

Există mai multe metode de a sorta topologic un dag. Prima are ca punct de plecare algoritmul de explorare DFS a unui graf. Reamintim că în timpul explorării DFS, un vârf poate fi întâlnit de mai multe ori. Notăm cu $f[v]$ momentul când vârful $v$ este întâlnit ultima dată (când lista de adiacenţă este epuizată). Algoritmul este descris schematic astfel:

- apelează algoritmul DFS pentru a determina momentele de terminare $f[v]$ pentru orice vârf $v$;
- se fiecare dată când un vârf este terminat este adăugat la începutul unei listei înlănţuite;
- lista înlănţuită finală va conţine o sortare topologică a vârfurilor.

Corectitudinea algoritmului se bazează pe următorul rezultat.

**Lema 4.1 ([CLR93])** *Un digraf $D$ este aciclic dacă şi numai dacă explorarea DFS nu produce arce înapoi.*

**Exerciţiul 4.4.1.** Să se scrie un program `sortareTopologicaDFS(D, S)` care implementează algoritmul de mai sus. Să se arate că algoritmul are timpul de execuţie $\Theta(\#V + \#A)$.

Cea de-a doua metodă pe care o studiem aici este o generalizare a explorării BFS. Presupunem că pentru dag-ul $D$ sunt create atât listele de adiacenţă interioară, cât şi cele de adiacenţă exterioară. Listele de adiacenţă interioară vor fi utilizate la determinarea surselor. Ca şi BFS, algoritmul de sortare topologică va utiliza o coadă.

- iniţializează coada cu vârfurile sursă (cele care au listele de adiacenţă interioară vide);
- extrage un vârf $u$ din coadă pe care-l adaugă la lista sortată parţial;
- elimină din reprezentarea (acum parţială) a lui $D$ vârful $u$ şi toate arcele $(u, v)$. Dacă pentru un astfel de arc lista de adiacenţă interioară a vârfului $v$ devine vidă, atunci $v$ va fi adăugat la coadă;
- repetă paşii 2 şi 3 până când coada devine vidă.

Extindem structura `D`, care reprezintă digraful $D$, cu tabloul `np[1..n]`: `D.np[u]` va conţine numărul succesorilor vârfului $u$. Subprogramul care implementează metoda de sortare topologică bazată pe explorărea BFS este următorul:

```
function sortareTopologicaBFS(D,np)
   coadaVida(C) //initializeaza coada C
   for u ← 0 to D.n-1 do
       // insereaza in C varfurile fara predecesor
       if D.np[u]=0 then insereaza(C,u)
   for k ← 0 to D.n-1 do
       // afiseaza varfurile in ordine topologica
       if EsteVida(C)
          then return ("Graful contine cicluri")
       u ← elimina(C)
       insereaza(L, lung(l), u)
```

```
            p ← D.a[u]
            while p≠NULL do
               v ← p->elt //v este un succesor imedial al lui u
               D.np[v] ← D.np[v]-1
               if D.np[v]=0
                  then insereaza(C,v)
               p ← p->succ
end
```

### 4.4.1 Exerciţii

**Exerciţiul 4.4.2.** Să se compare eficienţa algoritmului `sortareTopologicaBFS` cu cea a algoritmului `sortareTopologicaDFS`. De asemenea, să se compare ordinea vârfurilor în listele determinate de cei doi algoritmi.

**Exerciţiul 4.4.3.** Ce se întâmplă cu cei doi algoritmi dacă digraful de intrare are circuite?

## 4.5 Referinţe bibliografice

Acest capitol cuprinde numai o parte dintre algoritmii de sortare internă. Alţi doi algoritmi importanţi, sortarea prin interclasare şi sortarea rapidă, sunt prezentaţi în capitolul dedicat metodei *divide-et-impera*. O prezentare aproape completă (dar care trebuie actualizată) este [Knu76]. Algoritmii de sortare, datorită importanţei lor, sunt prezentaţi în toate cărţile dedicate studierii algoritmilor [Sed88, CLR93, CLR00, Baa78, BG00, Mel84a, AHU74, HS84, Hei96, KGGK94, MS91, LG86]. Această lucrare nu include algoritmi de sortare externă. Pentru cititorul interesat recomandăm [Knu76, Sed88, Baa78].

# Capitolul 5

# Căutare

Alături de sortare, căutarea în diferite structuri de date constituie una dintre operaţiile cele mai des utilizate în activitatea de programare. Problema căutării poate fi formulată în diverse moduri:

- dat un tablou $(\mathtt{s[i]} \mid 0 \leq i < n)$ şi un element $a$, să se decidă dacă există $i \in \{0, \ldots, n-1\}$ cu proprietatea $s[i] = a$;
- dată o structură înlănţuită (liniară, arbore, etc.) şi un element $a$, să se decidă dacă există un nod în structură a cărui informaţie este egală cu $a$;
- dat un fişier şi un element $a$, să se decidă dacă există o componentă a fişierului care este egală cu $a$ etc.

În plus, fiecare dintre aceste structuri poate avea sau nu anumite proprietăţi:

- informaţiile din componente sunt distincte două câte două sau nu;
- componentele sunt ordonate în conformitate cu o relaţie de ordine peste mulţimea informaţiilor sau nu;
- căutarea se poate face pentru toată informaţia memorată într-o componentă a structurii sau numai pentru o parte a sa numită *cheie*;
  - cheile pot fi unice (ele identifică în mod unic componentele) sau multiple (o cheie poate identifica mai multe componente);
- între oricare două căutări, structura de date nu suferă modificări (aspectul static) sau poate face obiectul operaţiilor de inserare/ştergere (aspectul dinamic).

Aici vom discuta numai o parte dintre aceste aspecte. Mai întâi le considerăm pe cele incluse în următoarea formulare abstractă:

> *Instanţă:* o mulţime univers $\mathcal{U}$, o submulţime $S \subseteq \mathcal{U}$ şi un element $a$ din $\mathcal{U}$;
> *Întrebare:* $x \in S$?

Aspectul *static* este dat de cazul când, între oricare două căutări, mulţimea $S$ nu suferă nici o modificare. Aspectul *dinamic* este obţinut atunci când între două căutări mulţimea $S$ poate face obiectul următoarelor operaţii:

- Inserare.
  *Intrare:* $S, x$;
  *Ieşire:* $S \cup \{x\}$.

| Tip de date | Implementare | Căutare | Inserare | Ştergere |
|---|---|---|---|---|
| Listă liniară | Tablouri | $O(n)$ | $O(1)$ | $O(n)$ |
| | Liste înlănţuite | $O(n)$ | $O(1)$ | $O(1)$ |
| Listă liniară ordonată | Tablouri | $O(\log_2 n)$ | $O(n)$ | $O(n)$ |
| | Liste înlănţuite | $O(n)$ | $O(n)$ | $O(1)$ |

Figura 5.1: Timpul de execuţie pentru cazul cel mai nefavorabil

- Ştergere.
  *Intrare:* $S, x$;
  *Ieşire:* $S \setminus \{x\}$.

Evident, realizarea eficientă a căutării şi, în cazul aspectului dinamic, a operaţiilor de inserare şi de ştergere, depinde de structura de date aleasă pentru reprezentarea mulţimii $S$.

## 5.1 Căutare în liste liniare

Mulţimea $S$ este reprezentată printr-o listă liniară. Dacă mulţimea $\mathcal{U}$ este total ordonată, atunci $S$ poate fi reprezentată de o listă liniară ordonată. Algoritmii corespunzători celor trei operaţii au fost deja prezentaţi în secţiunile 2.1, respectiv 2.2. Timpul de execuţie pentru cazul cel mai nefavorabil este dependent de implementarea listei. Tabelul 5.1 include un sumar al valorilor acestor timpi de execuţie. Facem observaţia că valorile pentru operaţiile de inserare şi ştergere nu presupun şi componenta de căutare. În mod obişnuit, un element $x$ este adăugat la $S$ numai dacă el nu apare în $S$; analog, un element $x$ este şters din $S$ numai dacă el apare în $S$. Deci ambele operaţii ar trebui precedate de căutare. În acest caz, la valorile timpilor de execuţie pentru inserare şi ştergere se adaugă şi valoarea corespunzătoare pentru căutare. De exemplu, timpul de execuţie în cazul cel mai nefavorabil pentru inserare în cazul în care $S$ este reprezentată prin tablouri neordonate devine $O(n)$,iar pentru cazul tablourilor ordonate rămâne aceeaşi, $O(n)$.

Pentru calculul timpului mediu de execuţie vom presupune că $a \in S$ cu probabilitatea $q$ şi că $a$ poate apărea în $S$ la adresa $adr$ cu aceeaşi probabilitate $\frac{q}{n}$. Timpul mediu de execuţie al căutărilor cu succes ($a$ este găsit în $S$) este:

$$T^{med,s}(n) = \frac{3q(1+2+\cdots+n)}{n} + 2q = \frac{3q(n+1)}{2} + 2q,$$

iar în cazul general avem:

$$T^{med}(n) = 3n - \frac{3nq}{2} + \frac{3q}{2} + 2.$$

**Cazul când se ia în considerare frecvenţa căutărilor.** Presupunem că $x_i$ este căutat cu frecvenţa $f_i$. Se poate demonstra că se obţine o comportare în medie bună, atunci când $f_1 \geq \cdots \geq f_n$. Dacă aceste frecvenţe nu se cunosc aprioric, se pot utiliza *tablourile cu autoorganizare*. Într-un tablou cu autoorganizare, ori

de câte ori se caută un $a = s[i]$, acesta este deplasat la începutul tablourilui în modul următor: elementele de pe pozițiile $1, \ldots, i-1$ sunt deplasate la dreapta cu o poziție după care se pune $a$ pe prima poziție. Dacă în loc de tablouri se utilizează liste înlănțuite, atunci deplasările la dreapta nu mai sunt necesare. Se poate arăta [Knu76] că pentru tablourile cu autoorganizare timpul mediu de execuție este:

$$T^{med,s}(n) \approx \frac{2n}{\log_2 n}$$

## 5.2 Modelul de calcul al arborilor de decizie pentru căutare

În această secțiune definim o clasă de algoritmi bazați pe paradigma *divide-et-impera* și arătăm că algoritmul de căutare binară este optim în această clasă.

Presupunem că mulțimea $S$ este reprezentată de tabloul $(\mathtt{s[i]} \mid 0 \leq i \leq n-1)$. Mai întâi generalizăm problema presupunând că se caută $a$ în secvența $(s[p], \ldots, s[q])$. Reamintim că are loc $s[p] < \cdots < s[q]$. Algoritmii de căutare bazați pe paradigma *divide-et-impera* [capitolul 10] au o descriere recursivă definită după următoarea strategie:

- se determină $m$ cu $p \leq m \leq q$;
- dacă $a = s[m]$, atunci căutarea se termină cu succes;
- dacă $a < s[m]$, atunci căutarea continuă cu subsecvența $(s[p], \ldots, s[m-1])$;
- dacă $a > s[m]$, atunci căutarea continuă cu subsecvența $(s[m+1], \ldots, s[q])$.

O schemă procedurală nerecursivă care descrie strategia de mai sus este următoarea:

```
function poz(s, n, a)
   1:  p ← 0      q ← n-1
   2:  alege m intre p si q
   3:  while ((a ≠ s[m]) and (p < q)) do
          4:  if (a < s[m])
                 then q ← m-1
                 else p ← m+1
          5:  alege m intre p si q
   6:  if (a = s[m])
          then return m
          else return -1
end
```

În funcție de modul de alegere a valorii $m$ prin instrucțiunile 2 și 5, se disting mai mulți algoritmi de căutare. Cei mai cunoscuți dintre aceștia sunt:

- *căutare liniară* (secvențială). Se alege $m = p$;
- *căutare binară*. Se alege $m = \lceil \frac{p+q}{2} \rceil$;
- *căutare Fibonacci*. Se presupune $q + 1 - p = Fib(k) - 1$, unde $Fib(k)$ este la $k$-lea număr Fibonacci. Se alege $m$ astfel încât $m - p = Fib(k-1) - 1$ și $q - m = Fib(k-2) - 1$.

Numărul de comparaţii pentru cazul cel mai nefavorabil este:

- căutare liniară: $O(n)$.
- căutare binară: $O(\log_2 n)$.
- căutare Fibonacci: $O(\log_2 n)$

Pentru a studia timpul mediu de execuţie şi criterii de optimalitate este necesar să precizăm formal modelul de calcul. Acesta corespunde arborilor de decizie pentru căutare definiţi de schema procedurală de mai sus.

**Definiţia 5.1.** Arborele de decizie pentru căutare *de dimensiune $n$ ataşat unui algoritm bazat pe metoda divide-et-impera este definit după cum urmează:*

- *mai întâi se defineşte recursiv arborele $T(p, q)$ astfel:*
  - *dacă $p > q$ atunci $T(p, q)$ este arborele vid;*
  - *altfel, rădăcina este $m$ calculată de algoritmul utilizat de instrucţiunea 2 sau 5, iar subarborele stâng este $T(p, m-1)$ şi cel drept este $T(m+1, q)$;*
- *arborele de decizie pentru căutare de dimensiune $n$ este $T(0, n-1)$ la care se adaugă vârfurile externe având ca etichete intervalele $(-\infty, X_0), (X_0, X_1), \ldots, (X_{n-1}, +\infty)$ în această ordine de la stânga la dreapta, unde $X_0, \ldots, X_{n-1}$ sunt $n$ variabile.*

Calculul *unui arbore de decizie pentru intrarea $x_0, \ldots, x_{n-1}, a$, unde $x_0 < \cdots < x_{n-1}$, constă în:*

1. *etichetarea nodurilor interne cu $x_0, \ldots, x_{n-1}$, astfel încât lista inordine ne dă ordinea crescătoare a etichetelor şi*
2. *parcurgerea unui drum de la rădăcină spre frontieră determinat astfel: dacă vârful curent $v$ este etichetat cu $x_m$ (vom vedea că $m$ este dat de instrucţiunea 2 sau 5 din schema procedurală divide-et-impera), atunci:*
   a) *dacă $v$ este vârf extern etichetat $(X_i, X_{i+1})$, atunci $a \in (x_i, x_{i+1})$ (variabila $X_i$ este interpretată ca având valoarea $x_i$) şi calculul se* termină cu insucces;
   b) *dacă $a = x_m$, atunci calculul se* termină cu succes;
   c) *dacă $a < x_m$, atunci rădăcina subarborelui stâng devine vârf curent;*
   d) *dacă $a > x_m$, atunci rădăcina subarborelui drept devine vârf curent.*

**Exemplu.** Arborele corespunzător căutării liniare pentru $n = 4$ este reprezentat în figura 5.2, iar cel corespunzător căutării binare pentru $n = 6$ este reprezentat în figura 5.3. sfex

Vârfurile de pe frontiera unui arbore de decizie pentru căutare se mai numesc şi *vârfuri pendante.* În continuare prezentăm câteva proprietăţi ale arborilor de decizie pentru căutare.

**Lema 5.1.** *Fie $t$ arborele de decizie pentru căutare cu $n$ vârfuri corespunzător căutării binare. Dacă $2^{h-1} \leq n < 2^h$, atunci înălţimea lui $t$ este $h$.*

*Demonstraţie.* Procedăm prin inducţie după $n$. Dacă $n = 1$, atunci afirmaţia din lemă este evident adevărată. Presupunem $n > 1$. Valoarea $m$ corespunzătoare rădăcinii este $\lceil \frac{n}{2} \rceil$. Din definiţia părţii întregi superioare rezultă următoarele inegalităţi:

$$2^{h-2} \leq m < 2^{h-1} + 1. \tag{5.1}$$

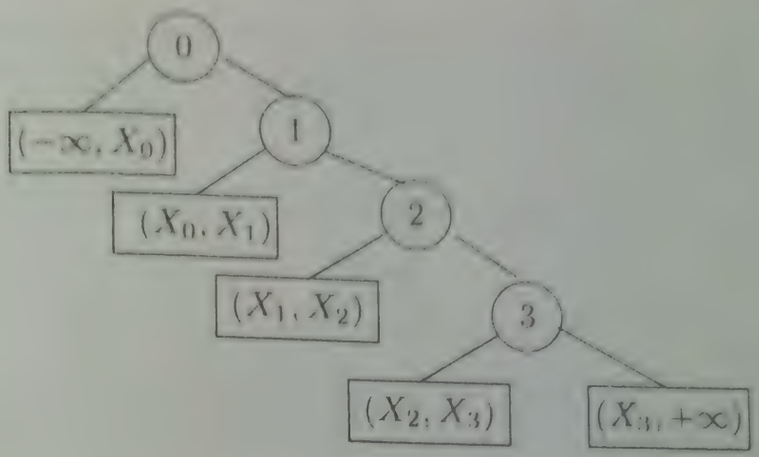



Figura 5.2: Arborele corespunzător căutării liniare

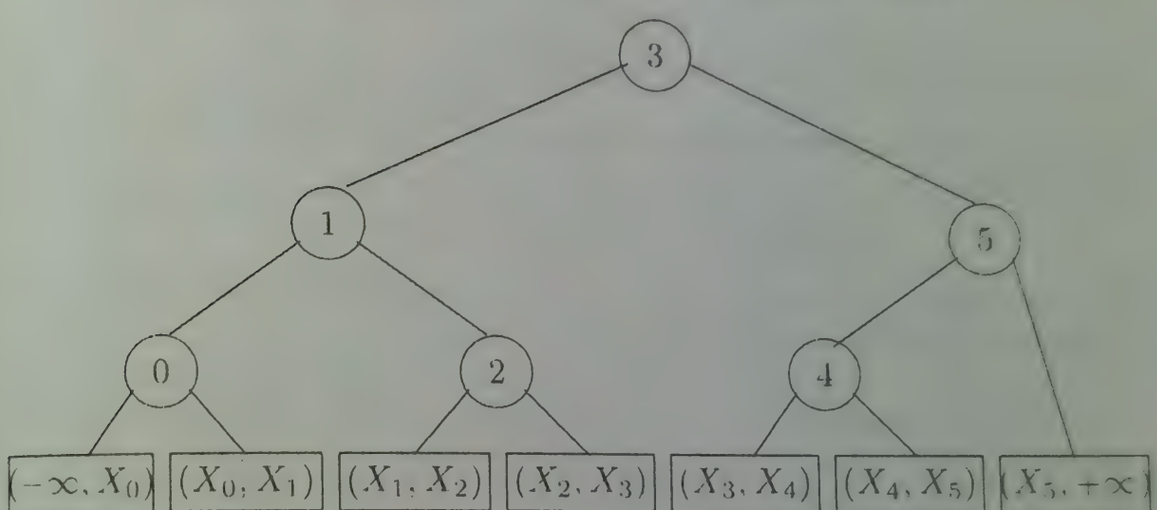



Figura 5.3: Arborele corespunzător căutării binare

Subarborii rădăcinii au $m$ şi respectiv $n-m-1$ vârfuri cu proprietatea $m-1 \leq n-m-1 \leq m$. Dacă $m = n-m-1$, atunci $m = \frac{n-1}{2}$. Deoarece $n \leq 2^h - 1$, rezultă că $m \leq 2^{h-1} - 1 < 2^{h-1}$. Dacă $m - 1 = n - m - 1$, atunci $m = \frac{n}{2} < 2^{h-1}$. Rezultă $m < 2^{h-1}$ în toate cazurile. Aplicând ipoteza inductivă, rezultă că subarborele cel mai înalt (cel cu $m$ vârfuri şi aflat la stânga rădăcinii) are înălţimea $h - 1$. Din definiţia înălţimii arborelui binar, rezultă că înălţimea lui $t$ este $h$. sfdem

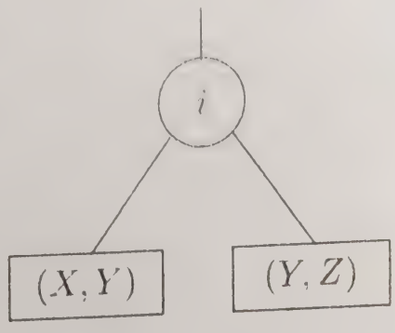
**Corolar 5.1.** *Timpul de execuţie pentru cazul cel mai nefavorabil al căutării binare este* $O(\log_2 n)$.

**Definiţia 5.2.** *Fie* $t$ *un arbore de decizie pentru căutare.* Lungimea internă *a lui* $t$, *notată* $\text{LungInt}(t)$, *este suma lungimilor drumurilor de la rădăcină la vârfurile interne.* Lungimea externă *a lui* $t$, *notată* $\text{LungExt}(t)$, *este suma lungimilor drumurilor de la rădăcină la vârfurile de pe frontieră (pendante).*

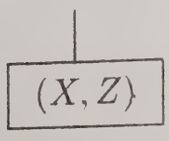
**Lema 5.2.** *Fie* $t$ *un arbore de decizie pentru căutare cu* $n$ *vârfuri interne. Atunci*

$$\text{LungExt}(t) - \text{LungInt}(t) = 2n.$$

*Demonstraţie.* Procedăm prin inducţie după $n$. Pentru $n = 1$ avem $\text{LungInt}(t) = 0$ şi $\text{LungExt}(t) = 2$. Presupunem $n > 1$. Fie $i$ un vârf intern cu ambii fii pe frontieră Înlocuim subarborele:




cu subarborele:




Noul arbore $t'$ este un arbore de decizie cu $n-1$ vârfuri interne şi conform ipotezei inductive avem

$$\text{LungExt}(t') - \text{LungInt}(t') = 2(n-1).$$

Deoarece $\text{LungExt}(t) = \text{LungExt}(t') + k + 2$ şi $\text{LungInt}(t) = \text{LungInt}(t') + k$, unde $k$ este lungimea drumului de la rădăcină la $i$, rezultă:

$$\begin{aligned} \text{LungExt}(t) - \text{LungInt}(t) &= \text{LungExt}(t') + k + 2 - (\text{LungInt}(t') + k) \\ &= 2n - 2 + k + 2 - k \\ &= 2n. \end{aligned}$$

sfdem

**Lema 5.3.** *Lungimea internă minimă a unui arbore de decizie cu $n$ vârfuri interne este:*

$$(n+1)(h-1) - 2^h + 2$$

*unde* $h = \lceil \log_2(n+1) \rceil$.

*Demonstraţie.* Considerăm formula:

$$x + x^2 + \cdots + x^k = \frac{x^{k+1} - x}{x - 1}.$$

Derivăm:

$$1 + 2x + \cdots + kx^{k-1} = \frac{kx^{k+1} - (k+1)x^k + 1}{(x-1)^2}.$$

Înmulţim cu $x$:

$$x + 2x^2 + \cdots + kx^k = x \cdot \frac{kx^{k+1} - (k+1)x^k + 1}{(x-1)^2}.$$

Luăm $k = h - 1$ şi $x = 2$:

$$2 + 2 \cdot 2^2 + \cdots + (h-1) \cdot 2^{h-1} = 2^h(h-2) + 2.$$

Lungimea internă minimă a unui arbore de decizie este suma primilor $n$ termeni din seria:

$$0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + \cdots$$

şi corespunde arborelui binar complet. Dacă presupunem $n+1 = 2^h$, atunci această sumă este egală cu:

$$2 + 2 \cdot 2^2 + \cdots + (h-1)2^{h-1} = 2^h(h-2) + 2.$$

Pentru cazul general trebuie să scădem suma drumurilor care unesc rădăcina cu vârfurile de pe nivelul $h$ care lipsesc:

$$2^h(h-2) + 2 - (2^h - 1 - n)(h-1) = (n+1)(h-1) - 2^h + 2.$$

sfdem

**Corolar 5.2.** *Lungimea externă minimă a unui arbore de decizie este:*

$$(n+1)(h+1) - 2^h.$$

*Demonstraţie.* Se aplică lemele precedente. sfdem

**Teorema 5.1.** *Problema căutării are timpul de execuţie în cazul cel mai nefavorabil* $\Omega(\log n)$ *în modelul arborilor de decizie pentru căutare.*

*Demonstraţie.* Arborele binar cu lungimea internă minimă considerat în lema 5.3 are şi înălţime minimă. În acest moment, concluzia teoremei rezultă din faptul că $2^{h-1} \leq n < 2^h$, unde $n$ este numărul de noduri, iar $h$ este înălţimea arborelui. sfdem

Presupunem că orice $x_i$ este căutat cu aceeaşi probabilitate $\frac{q}{n}$ şi că $a$ poate aparţine oricărui interval $(x_i, x_{i+1})$ (presupunem $x_0 = -\infty, x_{n+1} = \infty$) cu aceeaşi probabilitate $\frac{1-q}{n+1}$. Cu aceste ipoteze, timpul mediu de execuţie este dat de următoarea teoremă:

**Teorema 5.2.** *Fie $t$ un arbore de decizie pentru căutare. Timpul mediu de execuţie a căutărilor este:*

$$T_t^{med}(n) = \frac{2(n+q)}{n(n+1)} \cdot \mathrm{LungExt}(t) + 1 - 4q.$$

*Demonstrație.* O căutare constă în parcurgerea unui drum de la rădăcină spre frunză. Deoarece în fiecare nod se fac două comparații, iar la sfârșit încă o comparație, rezultă următoarea formulă pentru timpul mediu de execuție:

$$\begin{aligned} T_t^{med}(n) &= 1 + \frac{q}{n} \cdot 2 \cdot \text{LungInt}(t) + \frac{1-q}{n+1} \cdot 2 \cdot \text{LungExt}(t) \\ &= \dots \\ &= \frac{2(n+q)}{n(n+1)} \cdot \text{LungExt}(t) + 1 - 4q. \end{aligned}$$

sfdem

**Teorema 5.3.** *Căutarea binară este optimă din punctul de vedere al timpului mediu de execuție în modelul arborilor de decizie pentru căutare.*

*Demonstrație.* Se observă că arborele de decizie asociat căutării binare are lungimea externă minimă. sfdem

**Corolar 5.3.** *Timpul mediu de execuție pentru căutarea binară este* $\Theta(\log_2 n)$.

**Corolar 5.4.** *Problema căutării are timpul mediu de execuție* $\Omega(\log n)$ *în modelul arborilor de decizie pentru căutare.*

### 5.2.1 Căutare prin interpolare

Să presupunem că avem de căutat un cuvânt $w$ într-un dicționar. În general, procedăm astfel: deschidem dicționarul la întâmplare. Dacă observăm că pagina la care am deschis conține cuvinte mai mici decât $w$ (în ordinea lexicografică indusă de cea alfabetică), atunci vom sări un număr de pagini suficient de mare pentru a ne apropia de cuvântul căutat $w$. În cazul în care pagina deschisă conține cuvinte mai mari se sare un număr de pagini înapoi. Procedeul continuă până când se găsește pagina care îl conține pe $w$. Numărul de pagini sărit la fiecare pas depinde de distanța la care se află $w$ față de cuvintele de pe pagina curentă.

Metoda de căutare prin interpolare se bazează pe ideea de mai sus. Descrierea algoritmului este dată de schema de la metoda *divide-et-impera*, unde $m$ se determină prin formula:

$$m = (p-1) + \left\lceil \frac{a - s[p-1]}{s[q+1] - s[p-1]} (q+1-p) \right\rceil .$$

Este necesar să considerăm două elemente artificiale $s[-1]$ și $s[n]$ cu $s[-1] < s[0] \leq s[n-1] < s[n]$.

De remarcat că algoritmului de căutare prin interpolare nu-i putem atașa un arbore de decizie care să fie determinat numai de numărul $n$ de elemente din $S$, deoarece acesta depinde și de modul de dispersie al elementelor din $S$. De aceea, instrumentele prin care se analizează această metodă diferă de cele de la *divide-et-impera*. Se poate alege $S$ și $a$ astfel încât căutarea prin interpolare să degenereze în căutare liniară. De aici rezultă că are un timp de execuție de $O(n)$ în cazul cel mai nefavorabil. În ipoteza că elementele $x_0, \dots, x_{n-1}$ sunt repartizate uniform în intervalul $[x_{-1}, x_n]$ ($x_{-1} = s[-1]$, $x_n = s[n]$) rezultă un timp mediu de execuție $O(\log \log n)$ [Mel84a].

## 5.3 Arbori binari de căutare

Arborii $I(0, n-1)$ din definiţia arborilor de decizie pentru căutare sunt transformaţi în structuri de date înlănţuite asemănătoare cu cele definite în secţiunea 5.2. Aceste structuri pot fi definite într-o manieră independentă:

**Definiţia 5.3.** *Un* arbore binar de căutare *este un arbore binar cu proprietăţile:*

1. *informaţiile din noduri sunt elemente dintr-o mulţime total ordonată;*
2. *pentru fiecare nod $v$, valorile memorate în subarborele stâng sunt mai mici decât valoarea memorată în $v$, iar valorile memorate în subarborele drept sunt mai mari decât valoarea memorată în $v$.*

În continuare descriem implementările celor trei operaţii peste această structură.

Căutare. Operaţia de căutare într-un asemenea arbore este descrisă de următoarea procedură:

```
function poz(t, a)
   p ← t
   while ((p ≠ NULL) and (a ≠ p->elt)) do
   if (a < p->elt)
      then p ← p->stg
      else p ← p->drp
   return p
end
```

Funcţia poz ia valoarea $NULL$, dacă $a \notin S$ şi adresa nodului care conţine pe $a$ în caz contrar.

Operaţiile de inserare şi de ştergere trebuie să păstreze invariantă următoarea proprietate:

> *Valorile din lista inordine a nodurilor arborelui trebuie să fie în ordine crescătoare.*

Pentru a realiza operaţia de inserare se caută intervalul căruia îi aparţine $x$. Dacă în timpul procesului de căutare se găseşte un nod $*p$ cu $p\text{->}elt = x$, atunci arborele nu suferă nici o modificare (deoarece $x \in S$ implică $S \cup \{x\} = S$). Fie $p$ adresa nodului de pe frontieră care defineşte intervalul. Dacă $x < p\text{->}elt$ atunci $x$ se adaugă ca succesor la stânga; în caz contrar se adaugă ca succesor la dreapta. Un exemplu este arătat în figura 5.4.

Algoritmul care realizează operaţia de inserare are următoarea descriere:

```
procedure insereaza(t, x)
   if (t = NULL)
      then t ← nodNou(x)
      else q ← t
           while (q ≠ NULL) do
              p ← q
              if (x < q->elt)
                 then q ← q->stg
```

```
            else if (x > q->elt)
                    then q ← q->drp
                    else q ← NULL
        if (p->elt ≠ x)
           then q ← nodNou(x)
                if (x < p->elt)
                   then p->stg ← q
                   else p->drp ← q
end
```

Funcţia `nodNou()` creează un nod al arborelui binar şi întoarce adresa acestuia:

```
function nodNou(x)
   new(p)
   p->elt ← x
   p->stg ← NULL
   p->drp ← NULL
   return p
end
```

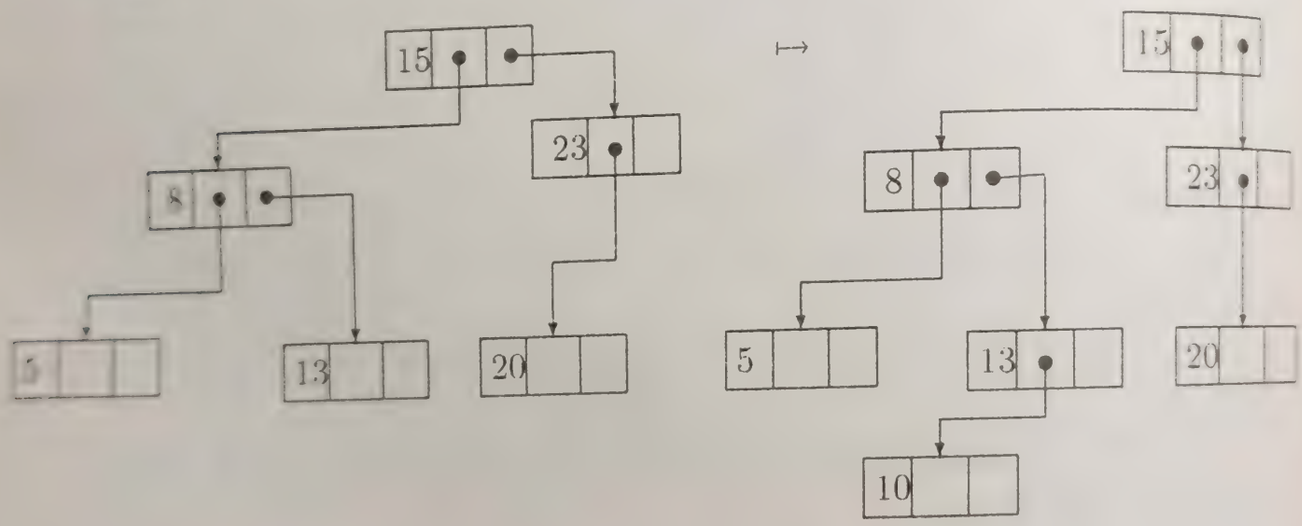



Figura 5.4: Inserare într-un arbore binar de căutare

Operaţia de ştergere se realizează într-un mod asemănător. Se caută $x$ în arborele care reprezintă mulţimea $S$. Dacă nu se găseşte un nod $*p$ cu $p\text{->}elt = x$, atunci arborele rămâne neschimbat (deoarece $x \notin S$ implică $S \setminus \{x\} = S$). Fie $p$ referinţa la nodul care conţine pe $x$. Se disting următoarele trei cazuri:

1. $*p$ nu are fii (figura 5.5.a). Se şterge nodul $*p$.
2. $*p$ are un singur fiu (figura 5.5.b). Părintele lui $*p$ este legat direct la fiul lui $*p$.
3. $*p$ are doi fii (figura 5.5.c). Se determină cea mai mare valoare dintre cele mai mici decât $x$. Fie aceasta $y$. Ea se găseşte memorată în ultimul nod $*q$ din lista inordine a subarborelui din stânga lui $*p$. Se transferă informaţia $y$ în nodul $*p$ după care se elimină nodul $*q$ ca în primele două cazuri.

Acum algoritmul de căutare are următoarea descriere:

```
procedure elimina(t, x)
   if (t ≠ NULL)
      then p ← t
           while ((p ≠ NULL) and (x ≠ p->elt)) do
              parintep ← p
              if (x < p->elt)
                 then p ← p->stg
                 else p ← p->drp
           if (p ≠ NULL)
              then if (p->stg = NULL) or (p->drp = NULL)
                      then elimCaz1sau2(parintep, p)
                      else q ← p->stg
                           parinteq ← p
                           while (q->drp ≠ NULL) do
                              parinteq ← q
                              q ← q->drp
                           p->elt ← q->elt
                           elimCaz1sau2(parinteq, q)
end

procedure elimCaz1sau2(parintep, p)
   if (p=t)
      then if (t->stg ≠ NULL)
              then t ← t->stg
              else t ← t->drp
      else if (p->stg ≠ NULL)
              then if (parintep->stg = p)
                      then parintep->stg ← p->stg
                      else parintep->drp ← p->stg
              else if (parintep->stg = p)
                      then parintep->stg ← p->drp
                      else parintep->drp ← p->drp
   delete(p)
end
```

De remarcat că ambele operaţii, inserarea şi eliminarea, necesită o etapă de căutare.

**Observaţie.** Structura de date utilizată aici se mai numeşte şi *arbore binar de căutare orientat pe noduri interne*. Această denumire vine de la faptul că elementele lui $S$ corespund nodurilor interne ale arborelui de căutare. Nodurile externe ale acestuia, care au ca informaţie intervalele deschise corespunzătoare căutărilor fără succes, nu au mai fost incluse în definiţia structurii de date din următoarele motive: simplitatea prezentării, economie de memorie şi, atunci când interesează, intervalele pot fi determinate în timpul procesului de căutare. Sugerăm cititorului, ca exerciţiu, să modifice algoritmul de căutare astfel încât, în cazul căutărilor fără succes, să ofere la ieşire intervalul deschis căruia îi aparţine $a$.

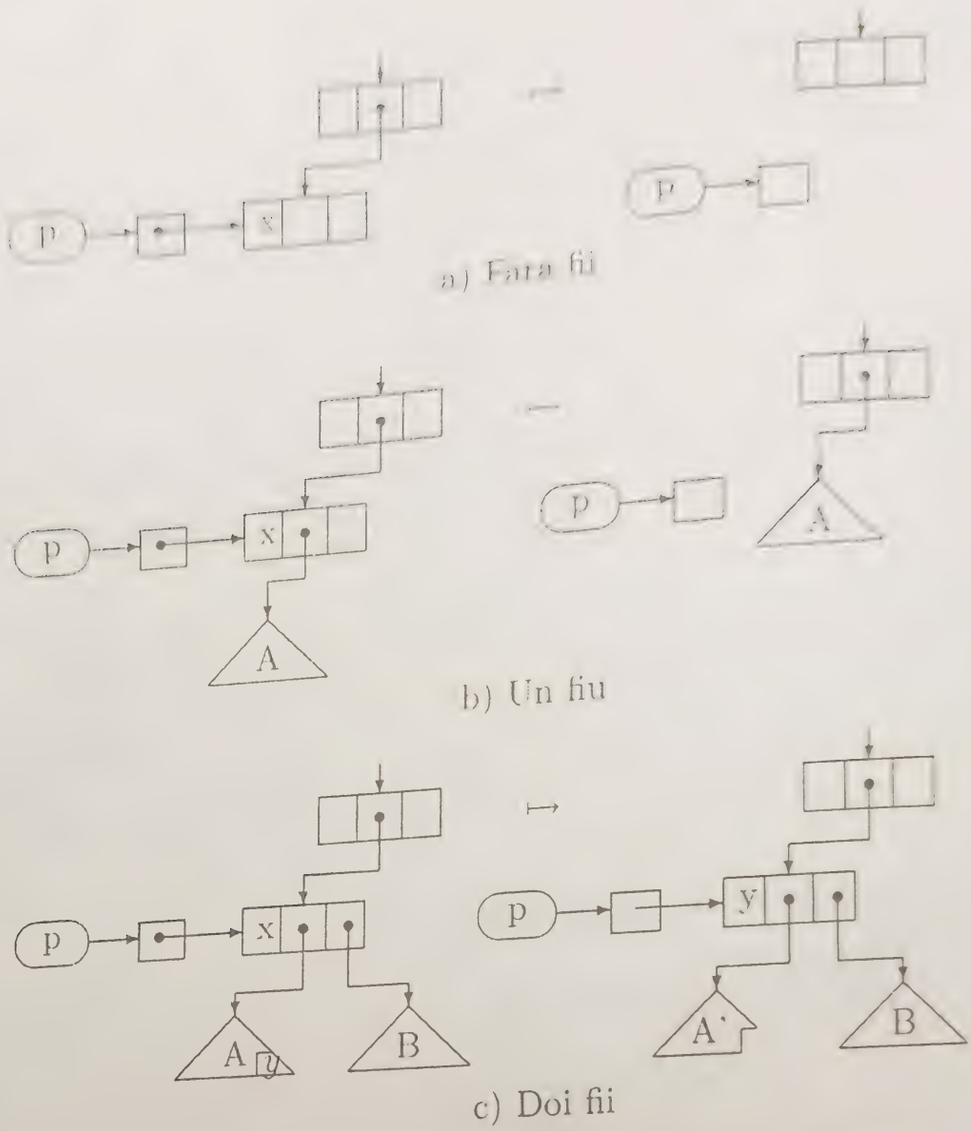



Figura 5.5: Ştergere dintr-un arbore binar de căutare

Mai există o variantă a structurii, numită *arbore binar de căutare orientat pe frontieră*, în care elementele lui $S$ sunt memorate atât în nodurile interne, cât şi în nodurile de pe frontieră. Informaţiile din nodurile de pe frontieră sunt în ordine crescătoare de la stânga la dreapta şi putem gândi că ele corespund intervalelor $(-\infty, x_0], (x_0, x_1], \ldots, (x_{n-1} + \infty)$. Algoritmul de căutare într-o astfel de structură va face testul $p\text{->}val = a$ numai dacă $*p$ este un nod pe frontieră. În cazul în care avem egalitate rezultă că $a$ aparţine mulţimii $S$; altfel, $a$ aparţine intervalului deschis mărginit la dreapta de $p\text{->}elt$. Pentru $S = \{5, 8, 13, 15, 20, 23\}$, structura de date este reprezentată schematic în figura 5.6. sfobs

**Exerciţiul 5.3.1.** Să se descrie proceduri pentru operaţiile de căutare, inserare şi ştergere pentru arbori binari de căutare orientaţi pe frontieră. Operaţiile vor păstra proprietatea ca fiecare nod să aibă exact doi fii.

### 5.3.1 Exerciţii

**Exerciţiul 5.3.2.** Un *nod complet* este un nod care are doi fii. Să se arate că într-un arbore binar numărul nodurilor complete este egal cu numărul nodurilor

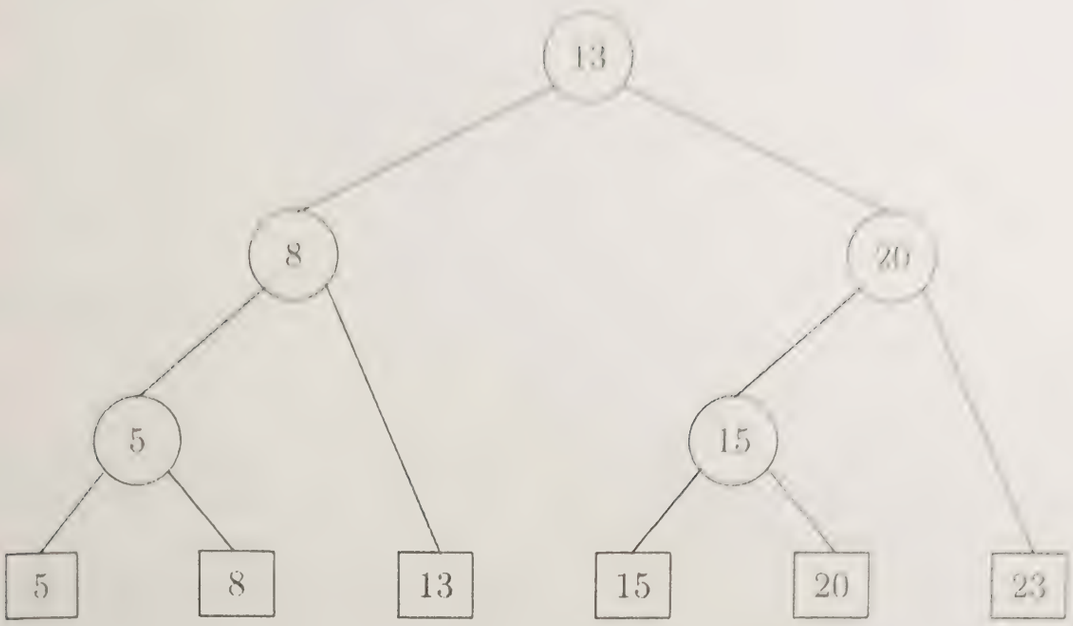



Figura 5.6: Arbore orientat pe frontieră

frunză minus 1.

**Exerciţiul 5.3.3.** Să se scrie un subprogram care determină cel mai mare element dintr-un arbore binar de căutare.

**Exerciţiul 5.3.4.** Să se scrie un subprogram care determină cel mai mic element dintr-un arbore binar de căutare.

**Exerciţiul 5.3.5.** Presupunem că se doreşte efectuarea unui experiment care să verifice problemele care apar la execuţiile aleatorii de perechi de operaţii inserare/ştergere. Următoarea strategie nu este perfect aleatoare dar e destul de aproape. Se construieşte un arbore cu $n$ elemente numere întregi, alese aleator din intervalul $[1, m]$, unde $m = \alpha \cdot n$. Apoi sunt realizate $n^2$ perechi de inserări urmate de ştergeri. Presupunem că există un subprogram **randInt**$(a, b)$ care întoarce un întreg ales aleator uniform din intervalul $[a, b]$.

1. Să se arate cum se generează aleator un număr întreg din intervalul $[1, m]$ care *nu* este deja în arbore. Ce se poate spune despre timpul de execuţie al acestei operaţii?
2. Să se arate cum se generează aleator un număr întreg din intervalul $[1, m]$ care este deja în arbore. Este utilă o astfel de operaţie? Ce se poate spune despre timpul de execuţie al acestei operaţii?
3. Care este o bună alegere pentru $\alpha$? De ce?

**Exerciţiul 5.3.6.** Să se scrie un subprogram care determină elementele $k$ dintr-un arbore binar de căutare cu proprietatea $k_1 \leq k \leq k_2$ pentru $k_1$ şi $k_2$ daţi. Care este timpul de execuţie al subprogramului?

# Capitolul 6

# Tipuri de date avansate pentru căutare

## 6.1 Arbori echilibraţi

Considerăm arbori binari de căutare. Este posibil ca în urma operaţiilor de inserare şi de ştergere structura arborelui binar de căutare să se modifice foarte mult şi operaţia de căutare să nu mai poată fi executată în timpul $O(\log_2 n)$. Un exemplu în care căutarea binară degenerează în căutare liniară este ilustrat în figura 6.1. Suntem interesaţi să găsim algoritmi pentru inserţie şi ştergere care să menţină o structură „echilibrată" a arborilor, astfel încât operaţia de căutare să se execute în timpul $O(\log n)$ întotdeauna. Mai întâi, definim formal astfel de clase. Menţionăm că în definiţiile următoare nu este necesar ca arborii să fie binari; ei pot fi de diferite arităţi.

Reamintim că înălţimea unui arbore $t$, pe care o notăm cu $h(t)$, este lungimea drumului maxim de la rădăcină la un vârf de pe frontieră.

**Definiţia 6.1.** *Fie $\mathcal{C}$ o mulţime de arbori. $\mathcal{C}$ se numeşte* clasă de arbori echilibraţi *(balansaţi), dacă pentru orice $t \in \mathcal{C}$, înălţimea lui $t$ este mai mică decât sau egală cu $c \cdot \log n$, unde $c$ este o constantă ce poate depinde de clasa $\mathcal{C}$ (dar nu depinde de $t$) şi $n$ este numărul de vârfuri din $t$.*

**Definiţia 6.2.** *O clasă $\mathcal{C}$ de arbori echilibraţi se numeşte* $O(\log n)$-stabilă *dacă există algoritmi pentru operaţiile de căutare, inserare şi de ştergere care necesită timpul $O(\log n)$ şi arborii rezultaţi în urma execuţiei acestor operaţii fac parte din clasa $\mathcal{C}$.*

În continuare prezentăm câteva clase $O(\log n)$-stabile.

### 6.1.1 Arbori AVL

Această clasă a fost definită de G.M. Adelson-Velskii şi E.M. Landis în 1962. În această subsecţiune ne referim la arbori binari de căutare orientaţi pe noduri interne.

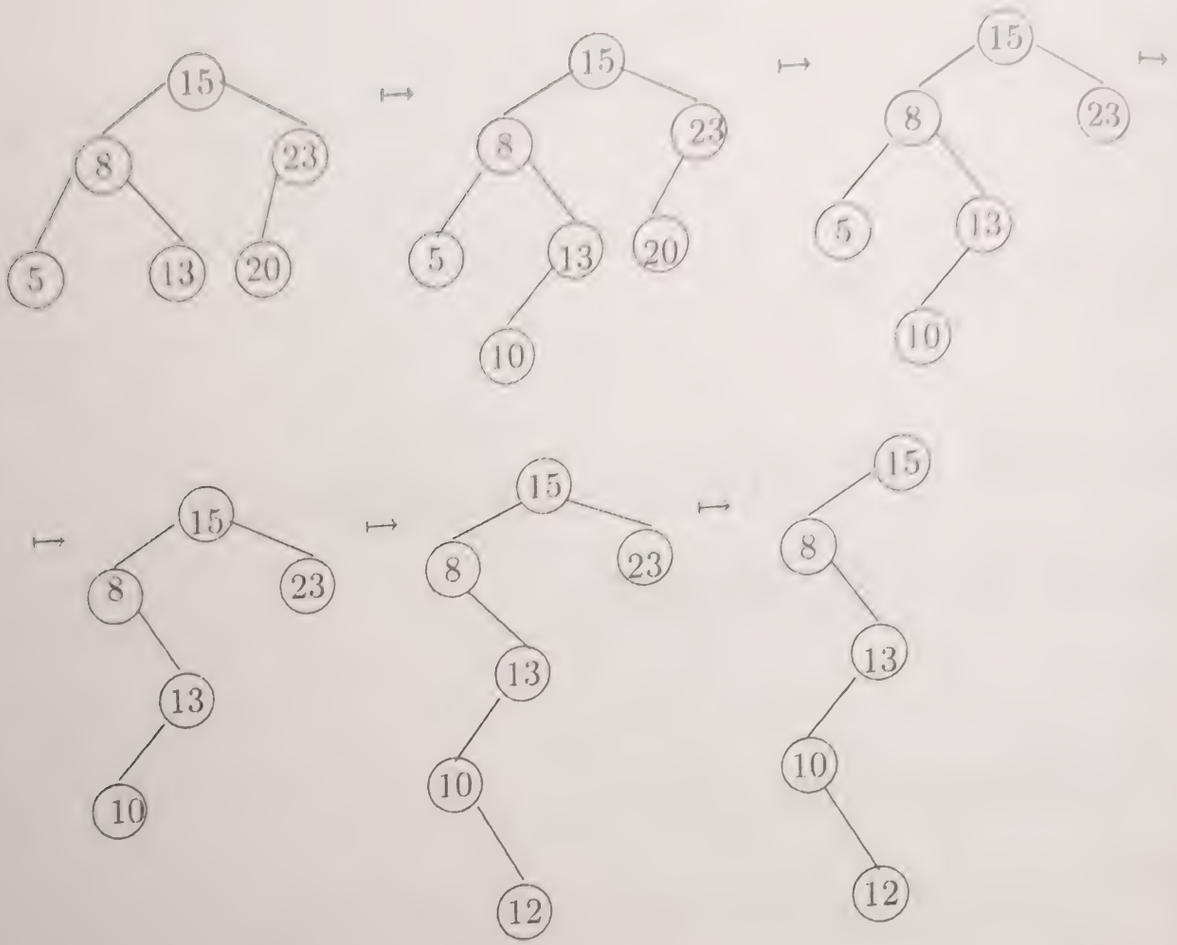



Figura 6.1: Degenerarea căutării binare în căutare liniară

Deşi nodurile externe ale arborelui de decizie, corespunzătoare intervalelor deschise, nu sunt incluse în structura de date, le considerăm la determinarea înălţimii (figura 6.2).

**Definiţia 6.3.** *Un arbore binar este* AVL-echilibrat *(pe scurt* arbore AVL*), dacă pentru orice vârf, diferenţa dintre înălţimea subarborelui din stânga vârfului şi cea a subarborelui din dreapta este* $-1$, $0$ *sau* $1$. *Numim această diferenţă* factor de echilibrare.

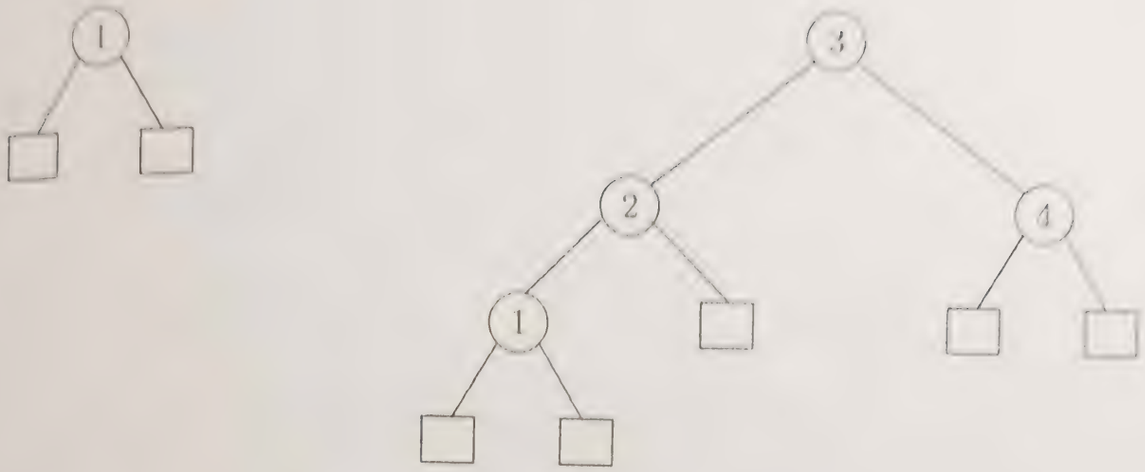
Următorul rezultat arată că arborii AVL sunt echilibraţi.

**Teorema 6.1.** *Pentru orice arbore AVL* $t$, *cu* $n$ *noduri interne, are loc* $h(t) = \Theta(\log_2 n)$.

*Demonstraţie.* Un arbore binar de înălţime $h$ are cel mult $2^h$ vârfuri pe frontieră şi cel mult $2^h - 1$ vârfuri interne. De aici rezultă:

$$h \geq log_2(n+1) = \Omega(\log_2 n) \tag{6.1}$$

Pentru a determina cea de-a doua margine, vom afla mai întâi structura unui arbore AVL de înălţime $h$, cu număr minim de noduri. Notăm acest arbore cu $t_h^m$. Procedăm prin inducţie după $h$. Dacă $h = 1$, atunci $t_1^m$ este format dintr-un singur nod intern corespunzător elementului $x_1$ şi două pe frontieră corespunzătoare intervalelor $(-\infty, x_0)$ şi $(x_0, +\infty)$. Dacă $h = 2$, atunci $t_2^m$ are două noduri interne




$h(t) = 1$ $h(t) = 3$

Figura 6.2: Calculul înălţimii arborilor AVL

corespunzătoare elementelor $x_0 < x_1$ şi trei vârfuri pe frontieră corespunzătoare intervalelor $(-\infty, x_0), (x_0, x_1), (x_1, +\infty)$. Remarcăm că $t_1^m$ are $Fib(3) - 1$ noduri interne iar $t_2^m$ are $Fib(4) - 1$ noduri interne, unde $Fib(k)$ este al $k$-lea număr Fibonacci. Presupunem $h > 2$. Presupunem că subarborele din stânga rădăcinii lui $t_h^m$ are înălţimea $h - 1$ şi subarborele din dreapta rădăcinii are înălţimea $h - 2$ (deoarece $t_h^m$ are număr minim de noduri, rezultă că nu pot avea ambii înălţimea $h - 1$). Deoarece $t_h^m$ are număr minim de noduri, rezultă că subarborii rădăcinii au număr minim de noduri. Fără să restrângem generalitatea putem presupune că aceşti subarbori sunt $t_{h-1}^m$ şi $t_{h-2}^m$. Rezultă că numărul de vârfuri interne ale lui $t_h^m$ este $Fib(h+1) - 1 + Fib(h) - 1 + 1 = Fib(h+2) - 1$. Deci $t_h^m$ coincide cu al $h+2$-lea arbore Fibonacci. Rezultă:

$$n \geq Fib(h+2) - 1 = \lceil \frac{\Phi^{h+2}}{\sqrt{5}} + \frac{1}{2} \rceil - 1 > \frac{\Phi^{h+2}}{\sqrt{5}} - \frac{3}{2} \tag{6.2}$$

unde $\Phi = \frac{1+\sqrt{5}}{2}$ („numărul de aur"). Din (6.2) obţinem:

$$h < \frac{1}{\log_2 \Phi} \cdot \log_2 \left( \sqrt{5}(n + \frac{3}{2}) \right) - 2 = O(\log_2 n). \tag{6.3}$$

Relaţiile (6.1) şi (6.3) implică concluzia din teoremă. sfdem

#### 6.1.1.1 Conservarea proprietăţii de arbore binar AVL-echilibrat

**Instrumente de reechilibrare – rotaţii.** În perspectiva conservării proprietăţii de arbore binar de căutare AVL-echilibrat, operaţiile `insereaza` şi `elemina` trebuie modificate, astfel încât după o astfel de operaţie arborele să rămână AVL-echilibrat. Adăugarea la operaţiile `insereaza` şi `elimina` a acestor mecanisme de conservare a tipului AVL a arborelui asupra căruia operează nu schimbă complexitatea acestora.

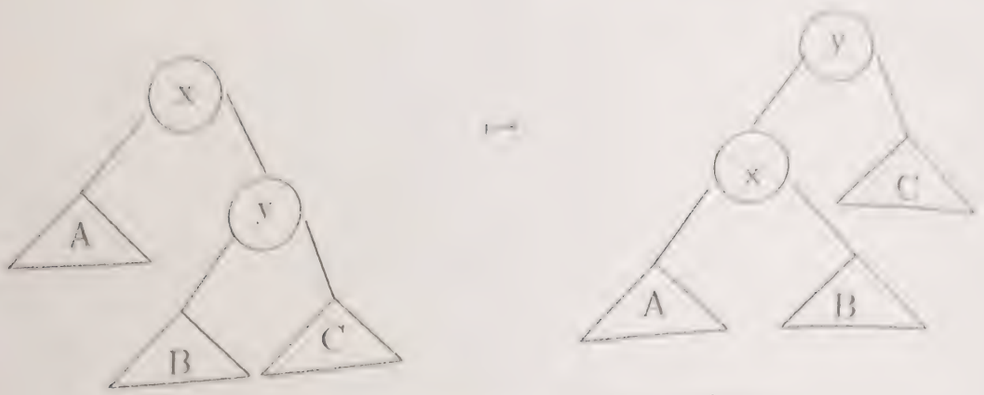



Figura 6.3: Rotaţie simplă la stânga

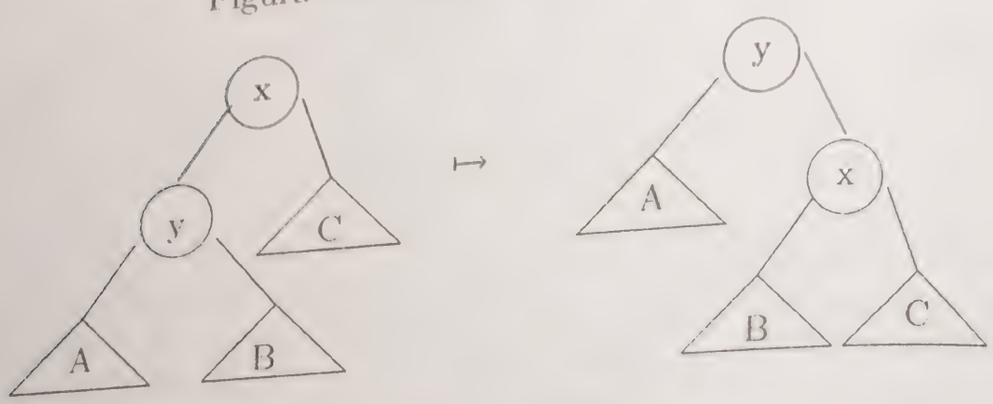



Figura 6.4: Rotaţie simplă la dreapta

Modificările necesare refacerii structurii de arbore binar echilibrat, după efectuarea unei operaţii de inserţie sau ştergere, se numesc *rotaţii*. Rotaţiile pot fi simple sau duble (figurile 6.3, 6.4, 6.5, 6.6).

Subpprogramele care implemenetează operaţiile implicate de rotaţii sunt următoarele:

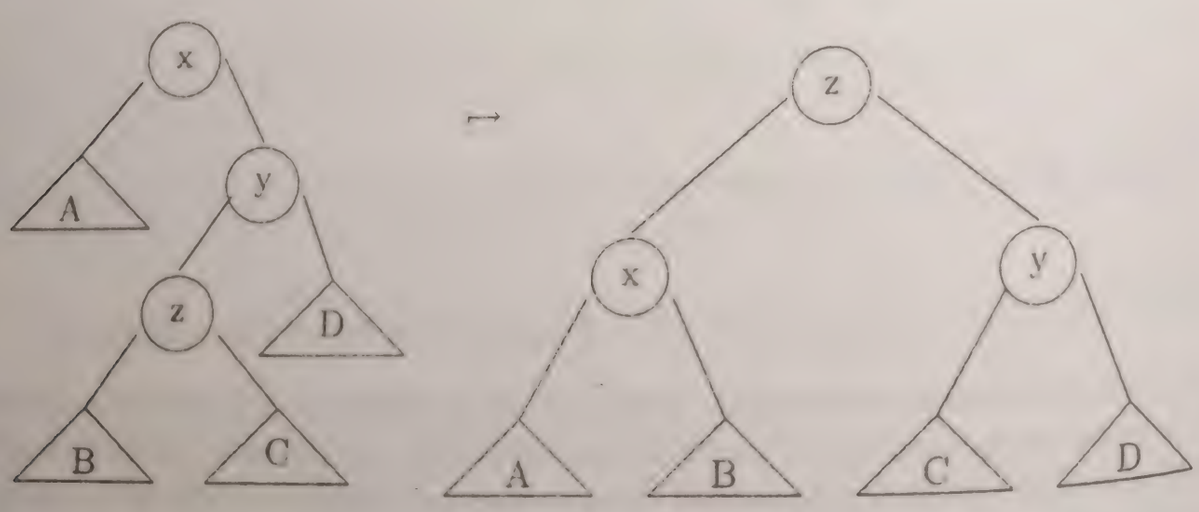



Figura 6.5: Rotaţie dublă la stânga

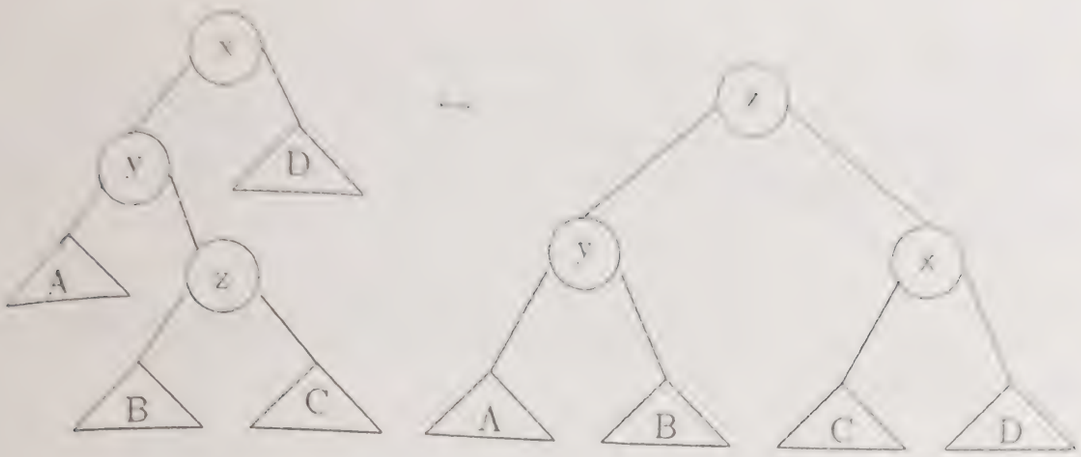



Figura 6.6: Rotaţie dublă a la dreapta

```
procedure rotatieSimplaStg(x)
   y ← x->drp
   x->drp ← y->stg
   y->stg ← x
   x ← y /* se schimba radacina */
end

procedure rotatieSimplaDrp(x)
   y ← x->stg
   x->stg ← y->drp
   y->drp ← x
   x ← y
end

procedure rotatieDublaStg(x)
   y ← x->drp
   z ← y->stg
   x->drp ← z->stg
   y->stg ← z->drp
   z->stg ← x
   z->drp ← y
   x ← z
end

procedure rotatieDublaDrp(x)
   y ← x->stg
   z ← y->drp
   x->stg ← z->drp
   y->drp ← z->stg
   z->drp ← x
   z->stg ← y
   x ← z
end
```

### 6.1.1.2 Tehnici de reechilibrare a arborilor AVL

**Inserarea.** După inserarea unui nod terminal într-un arbore AVL este necesară cel mult o rotaţie. Identificarea nodului asupra căruia trebuie aplicată rotaţia este problema-cheie.

**Definiţia 6.4.** *Se numeşte factor de echilibru al unui nod $v$ şi se notează cu $bf(v)$ diferenţa dintre înălţimea subarborelui stâng şi înălţimea subarborelui drept: $bf(v) = h(v->stg) - h(v->drp)$.*

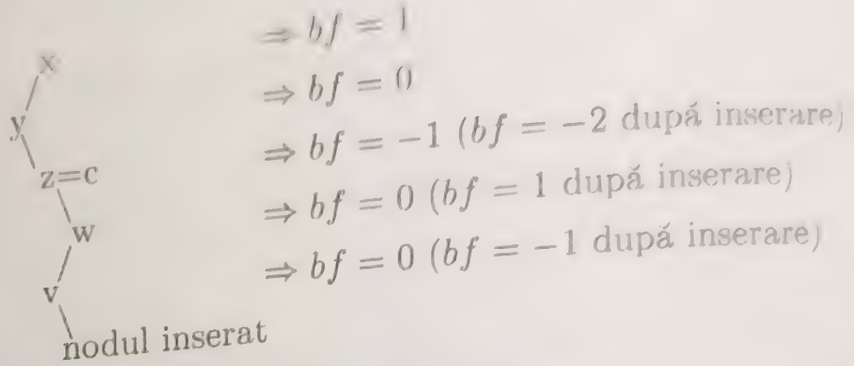



Figura 6.7: Determinarea nodului critic

**Observaţie.** După inserare, pentru nodul terminal $v$ şi nodul $x$ s-au schimbat doar înălţimea şi factorul de echilibru, aceştia rămânând totuşi echilibraţi. Nodul $z$ devine însă dezechilibrat, deoarece $bf(z)$ devine $-2$. sfobs

**Definiţia 6.5.** *Se numeşte nod critic primul nod $c$ cu $bf(c) \neq 0$ întâlnit la o parcurgere a ramurii care leagă nodul inserat de rădăcină, parcurgerea efectuându-se de la nod spre rădăcină. $bf(c)$ este cel calculat înainte de inserare.*

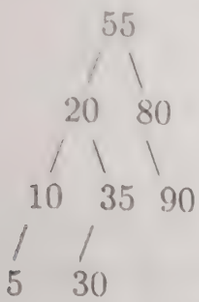
**Observaţie.** Toate nodurile din ramură care sunt pe nivele inferioare nodului critic vor avea după inserare $bf = 1$ sau $bf = -1$. sfobs

În cazul nodului critic $c$ există *două situaţii*:

1. Nodul critic va fi perfect echilibrat ($bf(c) = 0$), dacă dezechilibrul creat de nodul inserat anulează dezechilibrul iniţial al nodului. În acest caz, nu este nevoie de rotaţie (el completează un gol în arbore);
2. Factorul de echilibru devine $bf(c) = 2$ sau $bf(c) = -2$, atunci când nodul inserat măreşte dezechilibrul arborelui (s-a inserat nodul în subarborele cel mai înalt). În acest caz, se aplică o rotaţie în urma căreia se schimbă structura subarborelui cu rădăcina $c$, astfel încât pentru noua rădăcină $c'$, factorul de echilibru $bf(c')$ devine 0. Adâncimea subarborelui devine egală cu adâncimea acestuia înainte de inserare. Astfel, nodurile care se află pe drumul de la $c$ la rădăcina arborelui rămân cu factorul de echilibru identic cu cel dinainte de inserare.

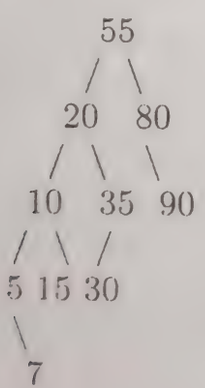
În concluzie, problema conservării proprietăţii de echilibru a unui arbore AVL în care s-a efectuat o inserare se rezolvă aplicând o rotaţie asupra nodului critic. Rotaţia se aplică *numai atunci* când inserarea dezechilibrează acest nod. Costul suplimentar care trebuie suportat rezultă din necesitatea ca în fiecare nod să se memoreze factorul de echilibru $bf$. Aceşti factori de echilibru trebuie actualizaţi la fiecare inserţie şi rotaţie.

**Exemplu.** Se dă arborele cu următoarea structură:

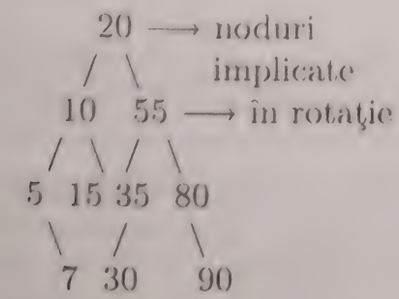



*Să se insereze nodurile 15, apoi 7 şi să se echilibreze arborele.*

Inserăm prima valoare 15. Comparăm mai întâi cu 55: e în stânga lui şi a.m.d. până când găsim locul ei în poziţia de mai jos. Pentru a doua valoare de inserat, 7, nodul critic este 55.

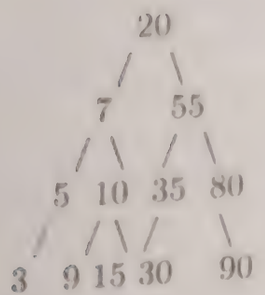



Este necesară aplicarea unei rotaţii asupra rădăcinii. Facem o identificare cu una din figurile de echilibrare prezentate anterior. Astfel 55→x; 20→y etc. Rezultă:




În urma unei rotaţii simple, factorii de echilibru implicaţi în rotaţie devin zero. Fie o a treia valoare de inserat, 3, apoi a patra, 9. Nodul critic pentru 3 este nodul de cheie 5. 3 echilibrează nodul critic, deci nu este nevoie de rotaţie. Nodul corespunzător lui 9 are nodul critic cel cu cheia 10 şi dezechilibrează subarborele

nodului critic. După o rotaţie dublă la dreapta aplicată acestui subarbore, rezultă arborele:




sfex

**Ştergerea.** Operaţia de ştergere într-un arbore de căutare AVL echilibrat implică mai multe rotaţii. Invităm cititorul să analizeze cazul ştergerii unui nod dintr-un arbore AVL-echilibrat.

**Teorema 6.2.** *Clasa arborilor AVL este $O(\log n)$-stabilă.*

*Demonstraţie.* În urma efectuării unei operaţii de inserare sau de ştergere singurele noduri care-şi modifică factorii de echilibrare se află pe drumul de la rădăcină la vârful inserat/şters. Se memorează acest drum într-o stivă în timpul etapei de căutare. După efectuarea operaţiei, se parcurge acest drum în sens invers şi se reface factorul de echilibru pentru nodurile dezechilibrate. Refacerea factorilor de echilibru se realizează cu ajutorul rotaţiilor. sfdem

## 6.1.2 Arbori bicolori

În această subsecţiune considerăm o clasă de arbori pentru căutare în care echilibrarea este menţinută cu ajutorul unei colorări adecvate ale nodurilor. De fapt, sunt suficiente numai două culori pentru a putea defini o clasă $O(\log n)$-stabilă. Deoarece aceste culori au fost iniţial roşu şi negru, arborii sunt cunoscuţi sub numele de *red-black-trees*.

**Definiţia 6.6.** *Un* arbore bicolor *este un arbore binar de căutare care satisface următoarele proprietăţi:*

1. *Nodurile sunt colorate. Un nod are sau culoarea roşie sau culoarea neagră.*
2. *Nodurile pendante (acestea sunt considerate ca făcând parte din structură) sunt colorate cu negru.*
3. *Dacă un nod este roşu, atunci fiii săi sunt coloraţi cu negru.*
4. *Pentru orice nod, drumurile de la acel nod la nodurile de pe frontieră au acelaşi număr de noduri colorate cu negru.*

Un exemplu de arbore bicolor este reprezentat în figura 6.8. Nodurile reprezentate cu cercuri albe reprezintă nodurile colorate cu roşu. Dacă $v$ este un nod într-un arbore bicolor $t$, atunci notăm cu $hn(v)$ numărul de noduri negre aflate pe un drum de la $v$ la un nod de pe frontieră, excluzând $v$. Definiţia lui $hn$ nu depinde de alegerea drumului datorită condiţiei 4 din definiţie.

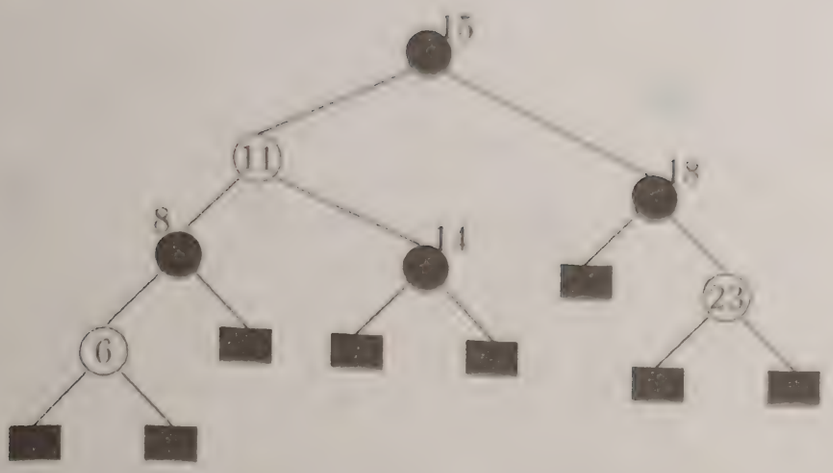



Figura 6.8: Arbore bicolor

**Lema 6.1.** *Fie $t$ un arbore bicolor. Orice subarbore al lui $t$ are cel puțin $2^{hn(v)} - 1$ noduri interne, unde $v$ este rădăcina subarborelui.*

*Demonstrație.* Procedăm prin inducție după înălțimea $h$ a subarborelui. Dacă $h = 0$, atunci $v$ este singurul nod al subarborelui și este un nod pendant. Numărul de noduri interne este 0 și coincide cu $2^{hn(v)} - 1 = 2^0 - 1$. Presupunem $h > 0$. Distingem două situații.

1. $v$ este roșu. Atunci fiii lui $v$ sunt negri. Fie $x$ și $y$ fiii lui $v$. Avem $hn(v) = hn(x) + 1 = hn(y) + 1$. Din ipoteza inductivă, fiecare subarbore al lui $v$ are cel puțin $2^{hn(v)-1} - 1$ noduri interne. Rezultă că $t$ are cel puțin $2 \cdot (2^{hn(v)-1} - 1) + 1 = 2^{hn(v)} - 1$ noduri interne.
2. $v$ este negru. Dacă $x$ și $y$ sunt ambii roșii atunci $hn(v) = hn(x) = hn(y)$. Din ipoteza inductivă, fiecare subarbore al lui $v$ are cel puțin $2^{hn(v)} - 1$ noduri interne. Rezultă că $t$ are cel puțin $2 \cdot (2^{hn(v)} - 1) + 1 > 2^{hn(v)} - 1$. Celelalte cazuri se tratează asemănător.

sfdem

**Teorema 6.3.** *Un arbore bicolor cu $n$ noduri interne are înălțimea cel mult $2\log_2(n+1)$.*

*Demonstrație.* Fie $t$ un arbore bicolor cu rădăcina $v$ și înălțimea $h$. Din lema 6.1 rezultă că $t$ are cel puțin $2^{hn(v)} - 1$ noduri interne. Din condiția 3 din definiția arborilor bicolori, rezultă că pe orice drum de la $v$ la un nod de pe frontieră cel puțin jumătate dintre noduri sunt negre. Avem $hn(v) \geq \frac{h}{2}$ care implică $n \geq 2^{hn(v)} - 1 \geq 2^{\frac{h}{2}} - 1$. De aici $h \leq 2\log_2(n+1)$.

sfdem

**Teorema 6.4.** *Clasa arborilor bicolori este $O(\log n)$-stabilă.*

*Demonstrație.* Din teorema 6.3 rezultă că operația de căutare, care se face exact ca la arborii de căutare obișnuiți, se realizează în timpul $O(\log n)$. Mai trebuie să arătăm că operațiile de inserare și de ștergere pot fi realizate în timpul $O(\log n)$ și în urma realizării acestor operații rezultă arbori bicolori.

sfdem

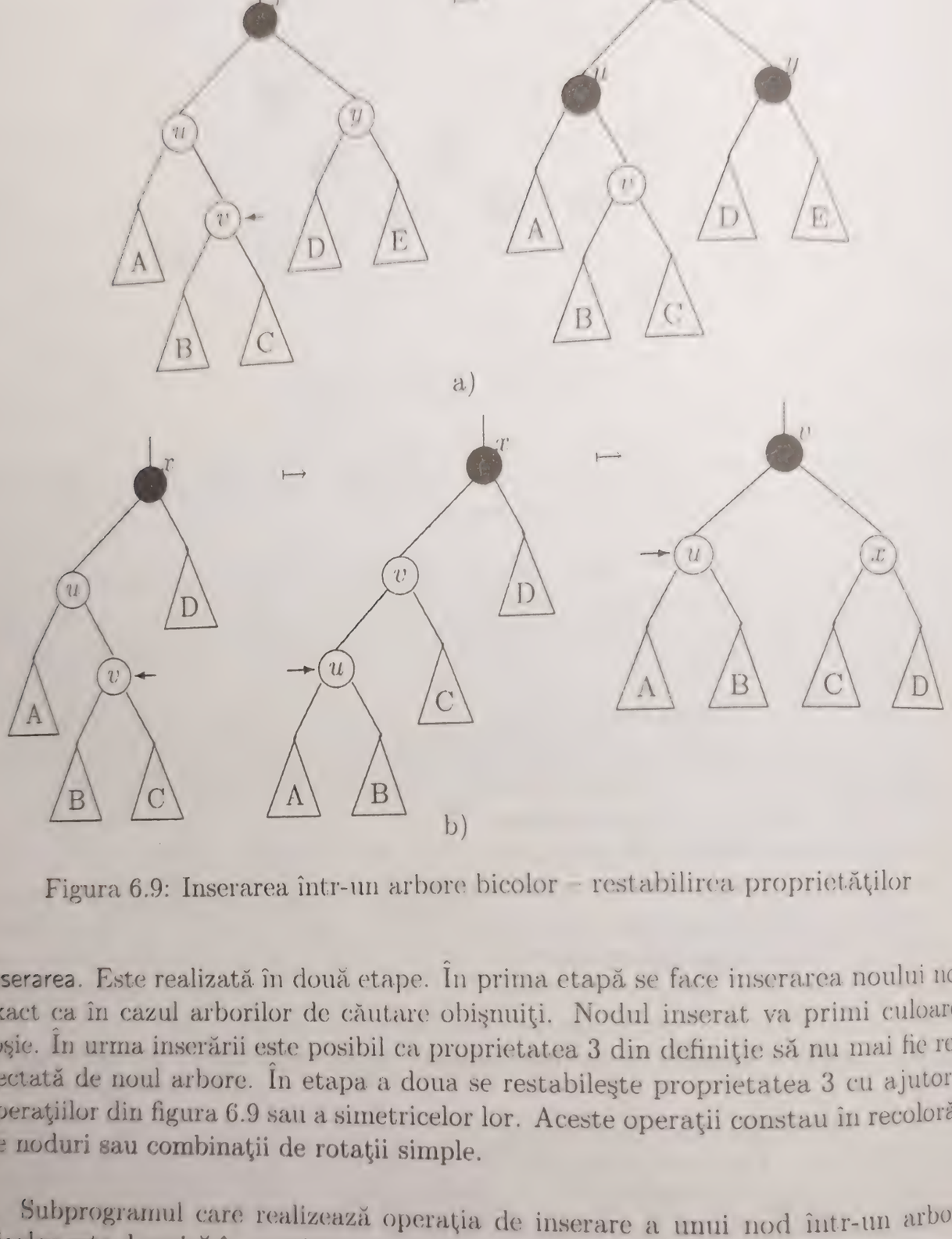



Figura 6.9: Inserarea într-un arbore bicolor – restabilirea proprietăţilor

Inserarea. Este realizată în două etape. În prima etapă se face inserarea noului nod exact ca în cazul arborilor de căutare obişnuiţi. Nodul inserat va primi culoarea roşie. În urma inserării este posibil ca proprietatea 3 din definiţie să nu mai fie respectată de noul arbore. În etapa a doua se restabileşte proprietatea 3 cu ajutorul operaţiilor din figura 6.9 sau a simetricelor lor. Aceste operaţii constau în recolorări de noduri sau combinaţii de rotaţii simple.

Subprogramul care realizează operaţia de inserare a unui nod într-un arbore bicolor este descrisă în continuare:

```
procedure insereazaBicolor(t, v)
   /* insereaza v in mod uzual (ca intr-un arbore binar de cautare) */
   insereaza(t, v)
   /* restabileste proprietatea de arbore bicolor */
   v->culoare ← rosu
   u ← v->parinte
   x ← u->parinte
```

```
while (v ≠ t si u->culoare = rosu)
  if ( u = x->stg )
     then y ← x->drept
          if ( y->culoare = rosu )
             then /* schimba culorile */
                  u->culoare ← negru
                  y->culoare ← negru
                  x->culoare ← rosu
                  /* v urca */
                  v ← x
             else /* y este negru */
                  if ( v = u->drp)
                     then /* v urca si rotatie */
                          v ← v->parinte
                          rotatieSimplaStg(v)
                  v->culoare ← negru
                  x->culoare ← rosu
                  rotatieSimplaDrp(x)
     else /* se repeta secventa de instructiuni de pe ramura then
             cu stanga si dreapta interschimbate */
t->culoare ← negru
end
```

Ştergerea. Şi această operaţie este realizată în două etape. Prima etapă seamănă cu cea de la arborii de căutare obişnuiţi, dar acum trebuie ţinut cont de faptul că nodurile pendante fac parte din structură. În urma ştergerii este posibil ca proprietatea 4 din definiţie să nu mai fie respectată de noul arbore. În partea a doua se restabileşte această proprietate cu ajutorul operaţiilor din figura 6.10. Nodurile reprezentate cu cercuri duble pot fi roşii sau negre. Cazul a) este transformat într-unul din cazurile b), c), d) printr-o rotaţie. În cazul b) nodul pentru care nu are loc proprietatea 4 este deplasat spre rădăcină cu un nivel prin recolorarea unui nod. Cazul c) este transformat în d) printr-o interschimbare de culori şi o rotaţie. Apariţia cazului d) şi rezolvarea lui conduce la restabilirea proprietăţii pentru întregul arbore.

Invităm cititorul să scrie subprograme pentru operaţia de ştergere.

### 6.1.3 2-3-arbori

2-3-arborii constituie o clasă de structuri de date arborescente $O(\log n)$-stabilă ce generalizează arborii binari de căutare.

**Definiţia 6.7.** *Un* 2-3-arbore *este un arbore de căutare care satisface următoarele proprietăţi:*

1. *Fiecare nod intern este de aritate 2 sau 3 (are 2 sau 3 fii). Un nod* $*v$ *de aritate 2 memorează o singură valoare* $v$->$eltStg$*, iar un nod de aritate 3 memorează două valori:* $v$->$eltStg$ *şi* $v$->$eltMij$ *cu* $v$->$eltStg < v$->$eltMij$.
2. *Fiu unui nod* $*v$ *îi vom numi fiu stânga (notat* $*(v$->$stg)$*), mijlociu (*$*(v$->$mij)$*) şi dreapta (*$*(v$->$drp)$*). Un nod de aritate 2 are fiii stânga şi mijlociu.*

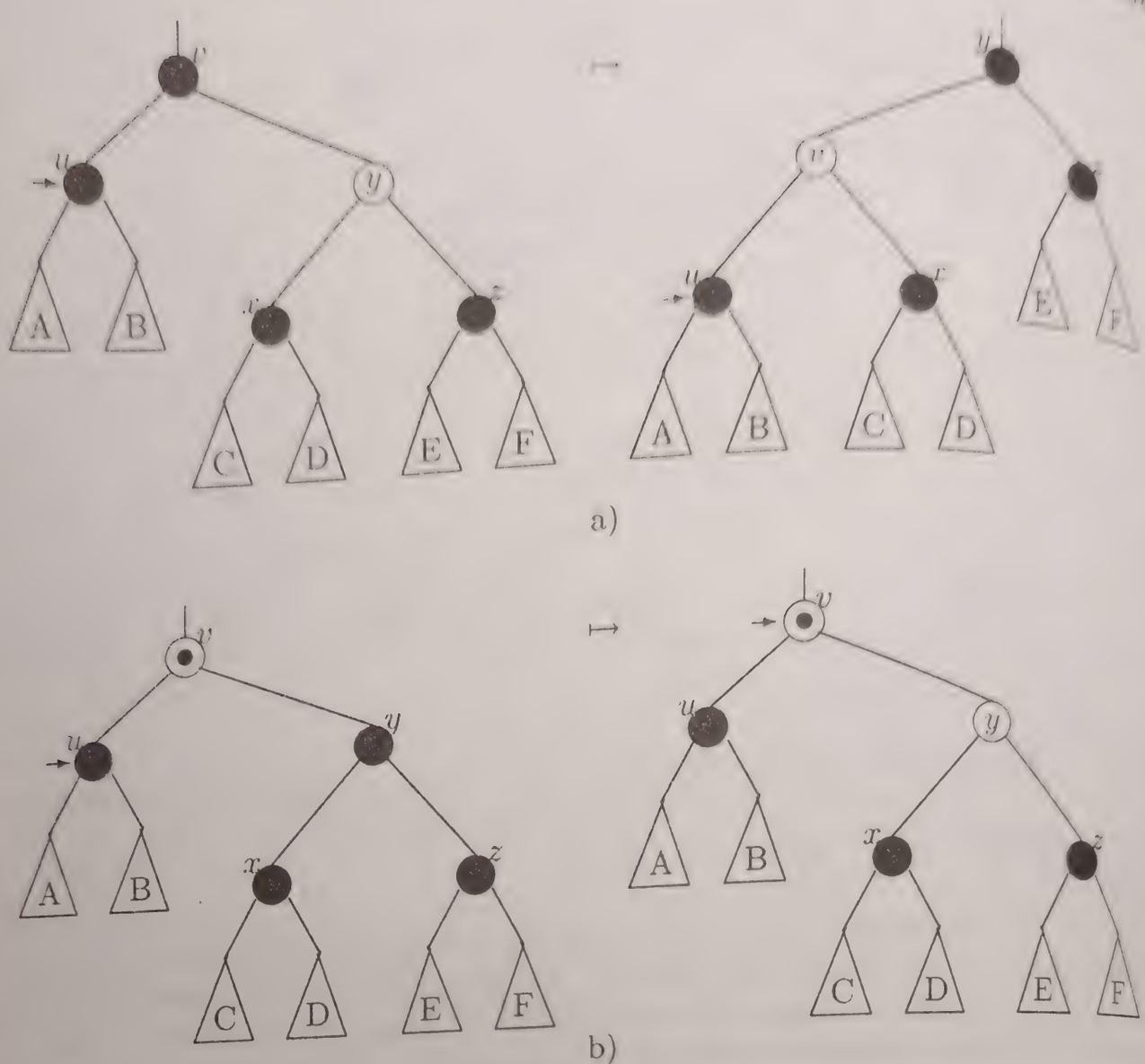



Figura 6.10: Ştergerea într-un arbore bicolor – restabilirea proprietăţilor

3. *Pentru orice nod intern $*v$ de aritate 2, valorile memorate în subarborele cu rădăcina în $*(v\text{->}stg)$ sunt mai mici decât $v\text{->}eltStg$ şi valorile din subarborele cu rădăcina în $*(v\text{->}mij)$ sunt mai mari decât $v\text{->}eltStg$.*
4. *Pentru orice nod intern $*v$ de aritate 3 are loc:*
   - *valorile din subarborele cu rădăcina $v\text{->}stg$ sunt mai mici decât $v\text{->}eltStg$;*
   - *valorile din subarborele cu rădăcina $v\text{->}mij$ sunt mai mari decât $v\text{->}eltStg$ şi mai mici decât $v\text{->}eltMij$;*
   - *valorile din subarborele cu rădăcina $*(v\text{->}drp)$ sunt mai mari decât $v\text{->}eltMij$.*
5. *Toate nodurile de pe frontieră se află pe acelaşi nivel.*

Un exemplu de 2-3-arbore este reprezentat grafic în figura 6.11. Nodurile reprezentate grafic prin pătrate nu intră în definiţia reprezentării, ele având rolul numai de a sugera mai bine structura de 2-3-arbore. Se poate stabili o corespondenţă între aceste noduri şi elementele mulţimii reprezentate. Facem presupunerea ca în cazul nodurilor de aritate 2, câmpul $v\text{->}eltMij$ să memoreze o valoare artificială $MaxVal$ mai mare decât orice element din mulţimea univers.

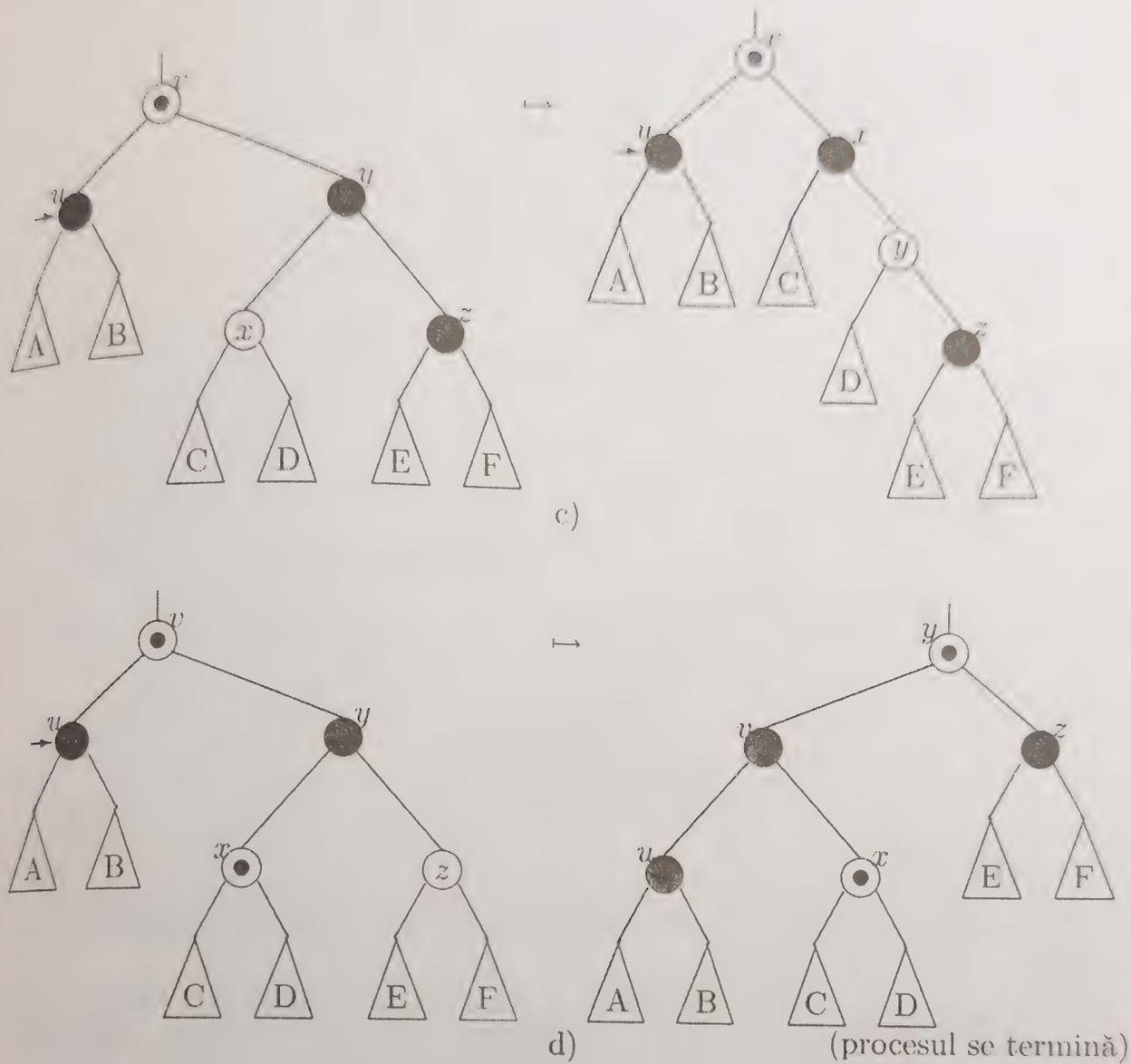



Figura 6.10: Ştergerea într-un arbore bicolor – restabilirea proprietăţilor (continuare)

Operaţia de căutare se realizează într-o manieră asemănătoare cu cea de la arborii binari de căutare. Procesul de căutare pentru $a$ în arborele $t$ pleacă din rădăcină şi în fiecare nod vizitat $*v$ se compară $a$ cu valorile memorate în $*v$:

- dacă $a = v\text{->}eltStg$ sau $a = v\text{->}eltMij$, atunci căutarea se termină cu succes;
- dacă subarborele în care se caută este vid, atunci căutarea se termină cu insucces;
- dacă $a < v\text{->}eltStg$, atunci procesul de căutare continuă în subarborele cu rădăcina în $v\text{->}stg$;
- dacă $v\text{->}eltStg < a < v\text{->}eltMij$, atunci procesul de căutare continuă în subarborele cu rădăcina în $v\text{->}mij$;
- dacă $a > v\text{->}eltMij$, atunci procesul de căutare continuă în subarborele cu rădăcina în $v\text{->}drp$.

Algoritmul care realizează toate acestea este următorul:

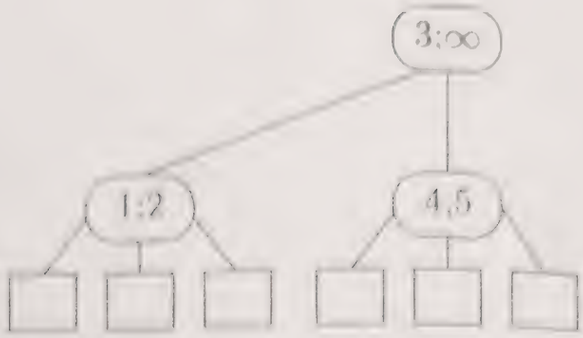



Figura 6.11: Exemplu de 2-3-arbore

```
function poz(t, a)
   p ← t
   while (p ≠ NULL)
      if ((p->eltStg = a) or (p->eltMij = a))
      then return p
      else if (a < p->eltStg)
           then p ← p->stg
           else if (a < p->eltMij)
                then p ← p->mij
                else p ← p->drp
   return p
end
```

Operaţia de inserare este ceva mai complicată şi se realizează după următorul scenariu:

1. Se caută în arborele $t$ elementul ce urmează a fi inserat $x$. Drumul de la rădăcină la frontieră parcurs în timpul căutării este memorat într-o stivă. Dacă $x$ este memorat deja în $t$ atunci operaţia de inserare se termină fără a modifica arborele.
2. În continuare se parcurge înapoi drumul memorat în stivă şi la fiecare pas se realizează o reuniune de doi arbori: unul care are rădăcina $y$ de aritate 1 şi subarborele lui $t$ cu rădăcina în nodul $*v$ memorat în vârful stivei. Inițial avem $y = x$ şi subarborele său vid. Întâlnim următoarele cazuri:
   a) Nodul $*v$ este de aritate 2. Atunci nodurile $y$ şi $*v$ fuzionează formând un singur nod de aritate 3 (figura 6.12). În acest caz, operaţia de inserare se termină.
   b) Nodul $*v$ este de aritate 3. Notăm cu $A$ subarborele lui $y$, cu $B, C, D$ subarborii lui $*v$ şi cu $x1, x2$ valorile memorate în $*v$. Pentru simplitatea prezentării notăm cei doi subarbori prin $[y](A)$ respectiv prin $[x1; x2](B, C, D)$. Nodul $*v$ se multiplică în două noduri de aritate 2 ale căror roluri în continuare diferă pentru fiecare din următoarele subcazuri:
      - $y < x1$. În funcţie de ce fel de fiu este $*v$, se formează următoarele perechi de subarbori (figura 6.13):
        - fiu stânga: $[x1]([y](A, B))$ şi $[x2](C, D)$.
        - fiu mijlociu sau dreapta: $[y](A, B)$ şi $[x1]([x2](C, D))$.

        În continuare se consideră $y = x1$.

- $y > x2$. Similar cazului precedent.
- $x1 < y < x2$. În funcţie de ce fel de fiu este la rândul său tatăl lui $*v$, se formează următoarele perechi de subarbori (figura 6.14)
  - fiu stânga: $[y]([x1](B, C))$ şi $[x2](A, D)$.
  - fiu mijlociu sau dreapta: $[x1](B, C)$ şi $[y]([x2](A, D))$.

  În continuare, $y$ rămâne neschimbat.

Dacă $v = t$ ($*v$ este rădăcină), atunci se crează un nou nod care va avea ca fii pe $y$ şi $*v$.

Ştergerea se poate realiza prin următoarea strategie:

1. Se caută în arborele $t$ elementul ce urmează a fi şters $x$. Drumul de la rădăcină la frontieră parcurs în timpul căutării este memorat într-o stivă. Dacă $x$ nu este în $t$, atunci operaţia de ştergere se termină fără a modifica arborele.
2. Fie $*v$ nodul unde este memorat $x$. Distingem situaţiile:
   a) $*v$ nod pe frontieră (figura 6.15).
      i) $*v$ are aritate 3: dacă $x = v\text{->}eltStg$, atunci se transferă valoarea din $v\text{->}eltMij$ în $v\text{->}eltStg$ şi se pune $MaxVal$ în $v\text{->}eltMij$, iar dacă $x = v\text{->}eltMij$ atunci se memorează $MaxVal$ în $v\text{->}eltMij$.
      ii) $*v$ de aritate 2. Putem presupune că după ştergere $*v$ devine de aritate 1 (are numai subarborele stânga). Acest tip de nod va fi „deplasat" pe drumul înapoi spre rădăcină până când este unit cu un frate de aritate 2 sau va avea tatăl de aritate 3 ori devine rădăcină. Deplasarea spre rădăcină a lui $v$ se face aplicând operaţii de tipul celor din figura 6.17. Dacă tatăl lui $*v$ are aritatea 3 atunci se aplică operaţia din figura 6.18 şi operaţia se termină. La fel, dacă la un moment dat $*v$ are un frate de aritate 3, atunci se aplică operaţia din figura 6.16 şi operaţia se termină. Dacă nodul $*v$ de aritate 1 ajunge rădăcină atunci se şterge.
   b) $*v$ este nod intern. Dacă $x = v\text{->}eltStg$, atunci se înlocuieşte $x$ cu cea mai mare valoare din subarborele cu rădăcina în $v\text{->}stg$ (sau cea mai mică valoare din subarborele cu rădăcina în $v\text{->}mij$), iar dacă $x = v\text{->}eltMij$, atunci se înlocuieşte $x$ cu cea mai mare valoare din subarborele cu rădăcina în $v\text{->}mij$ (sau cea mai mică valoare din subarborele cu rădăcina în $v\text{->}drp$). Valoarea care îl va înlocui pe $x$ este memorată într-un nod de pe frontieră şi, după înlocuire, va fi eliminată ca în primul caz.

**Teorema 6.5 ([Cro92])** *Clasa 2-3-arborilor este* $O(\log n)$ *stabilă.*

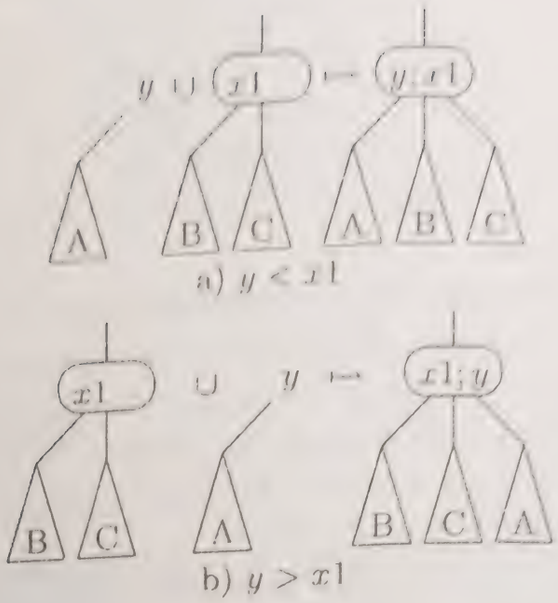



Figura 6.12: Inserare într-un 2-3-arbore: cazul când $v$ are aritatea 2

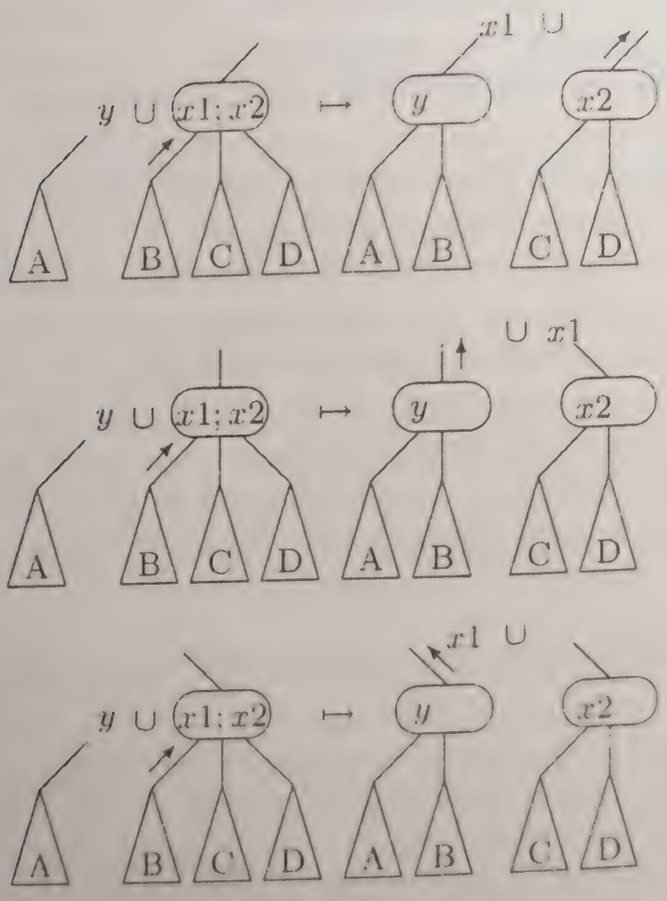



Figura 6.13: Inserare într-un 2-3-arbore: $v$ de aritate 3 și $y < v$->*cltStg*

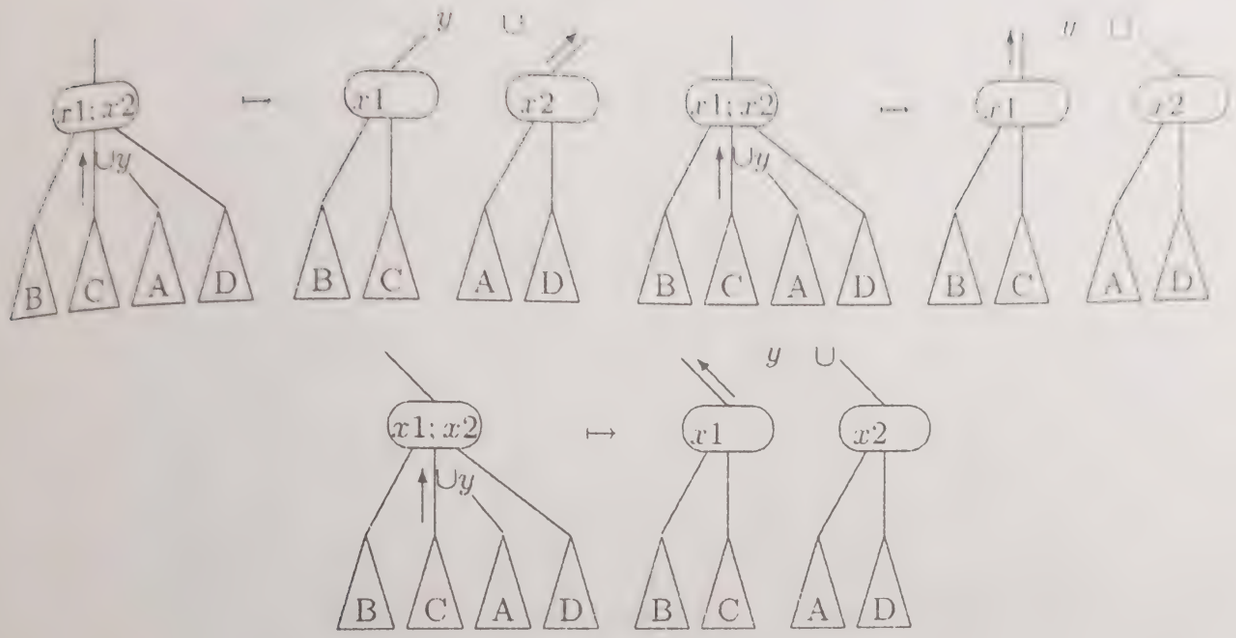



Figura 6.14: Inserare într-un 2-3-arbore: $v$ de aritate 3 şi $v$->$eltStg < y < v$->$eltMij$

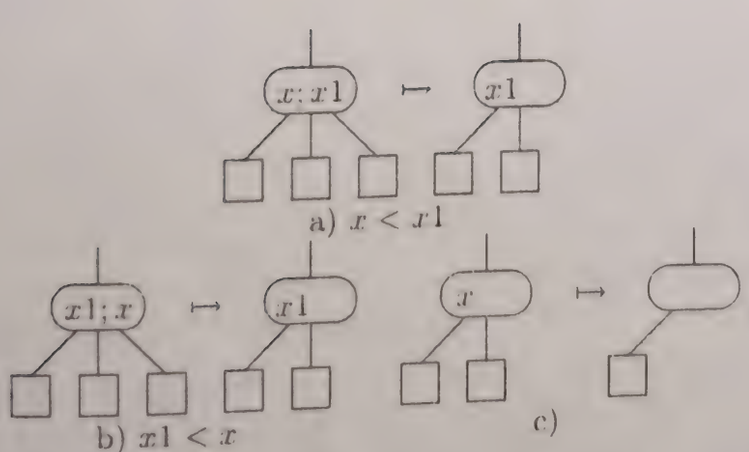



Figura 6.15: Ştergere într-un 2-3-arbore: $x$ este memorat într-un nod pe frontieră

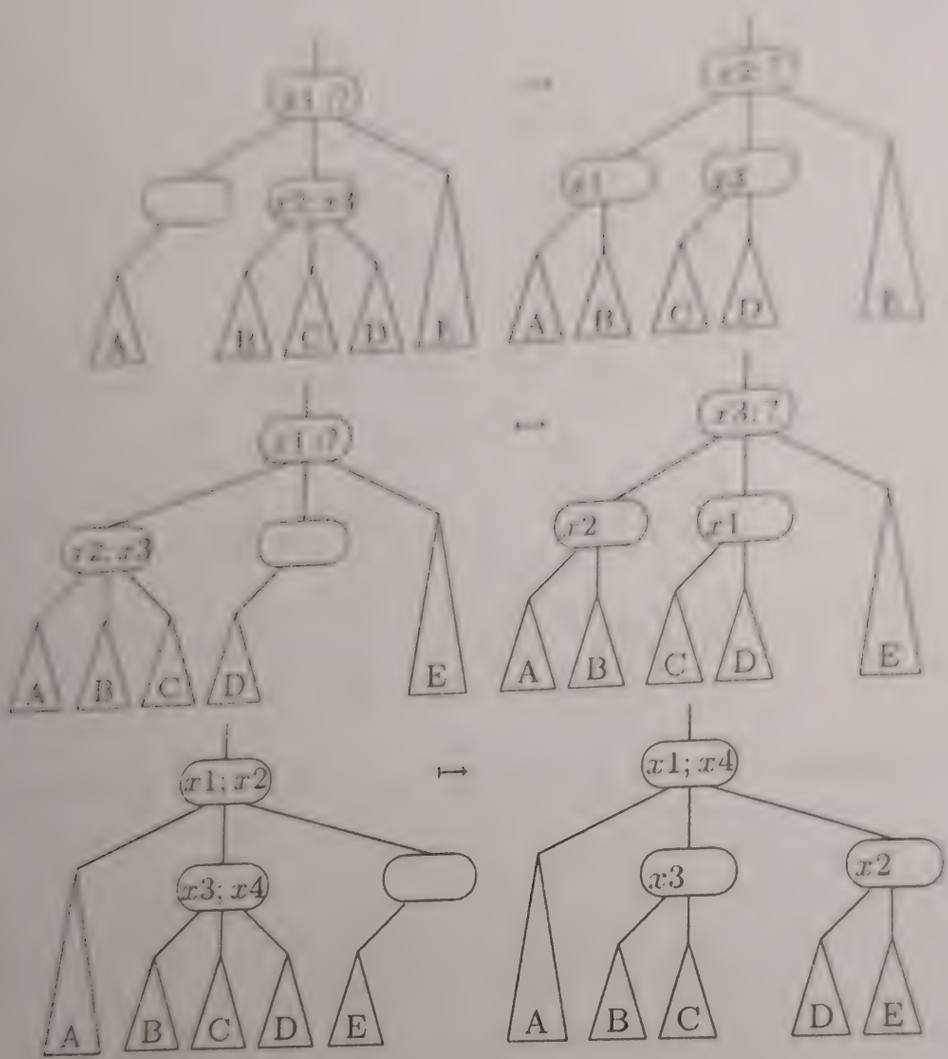



Figura 6.16: Ştergere într-un 2-3-arbore: rotaţii când $v$ are aritatea 3

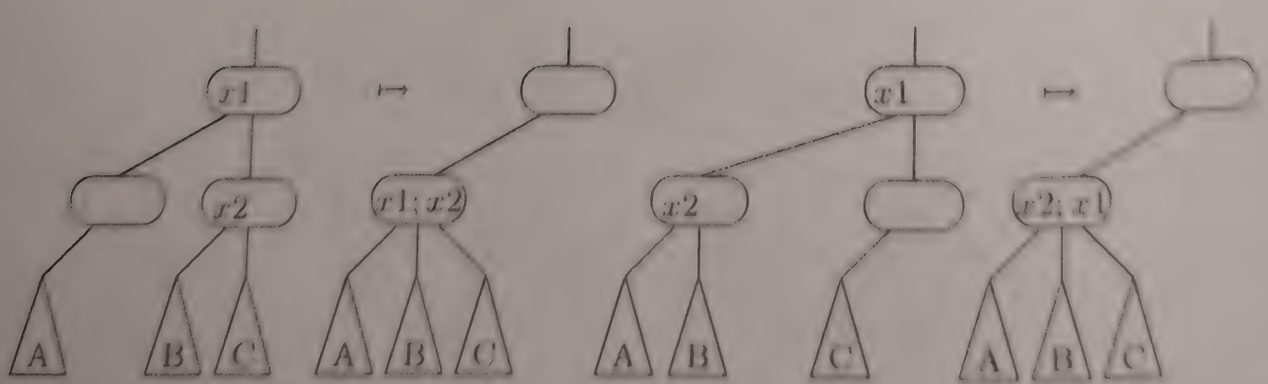



Figura 6.17: Ştergere într-un 2-3-arbore: combinarea nodurilor când $v$ şi tatăl său au aritatea 2

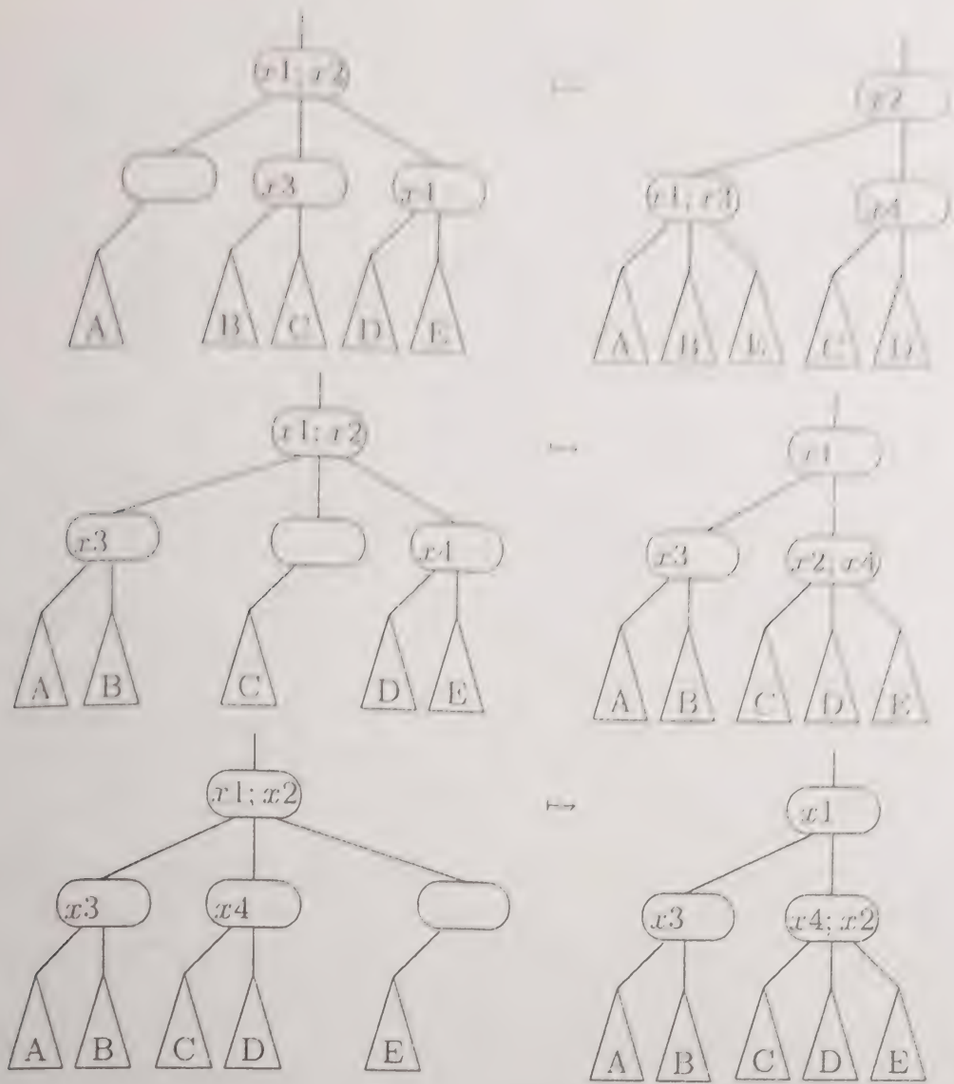



Figura 6.18: Ştergere într-un 2-3-arbore: combinarea nodurilor când $v$ are aritatea 2, iar tatăl său aritatea 3

### 6.1.4 B-arbori

B-arborii sunt frecvent utilizaţi la indexarea unei colecţii de date. Un index ordonat este un fişier secvenţial. Pe măsură ce dimensiunea acestuia creşte, cresc şi dificultăţile de administrare. Soluţia este construirea unui index cu mai multe nivele, iar instrumentele sunt B-arborii. Algoritmii de căutare într-un index nestratificat nu pot depăşi performanţa $O(\log_2 n)$ intrări/ieşiri. Indexarea multistratificată (vezi figura 6.19) are ca rezultat algoritmi de căutare de ordinul $O(\log_d n)$ intrări/ieşiri, unde $d$ este dimensiunea elementului indexului.

**Definiţia 6.8.** *Un B-arbore este un arbore cu rădăcină, în care:*

1. *fiecare nod intern are un număr variabil de chei şi fii;*
2. *cheile dintr-un nod intern sunt memorate în ordine crescătoare;*
3. *fiecare cheie dintr-un nod intern are asociat un fiu care este rădăcina unui subarbore ce conţine toate nodurile cu chei mai mici sau egale cu cheia respectivă dar mai mari decât cheia precedentă;*
4. *fiecare nod intern are un fiu extrem-dreapta, care este rădăcina unui subarbore ce conţine toate nodurile cu chei mai mari decât oricare cheie din nod;*

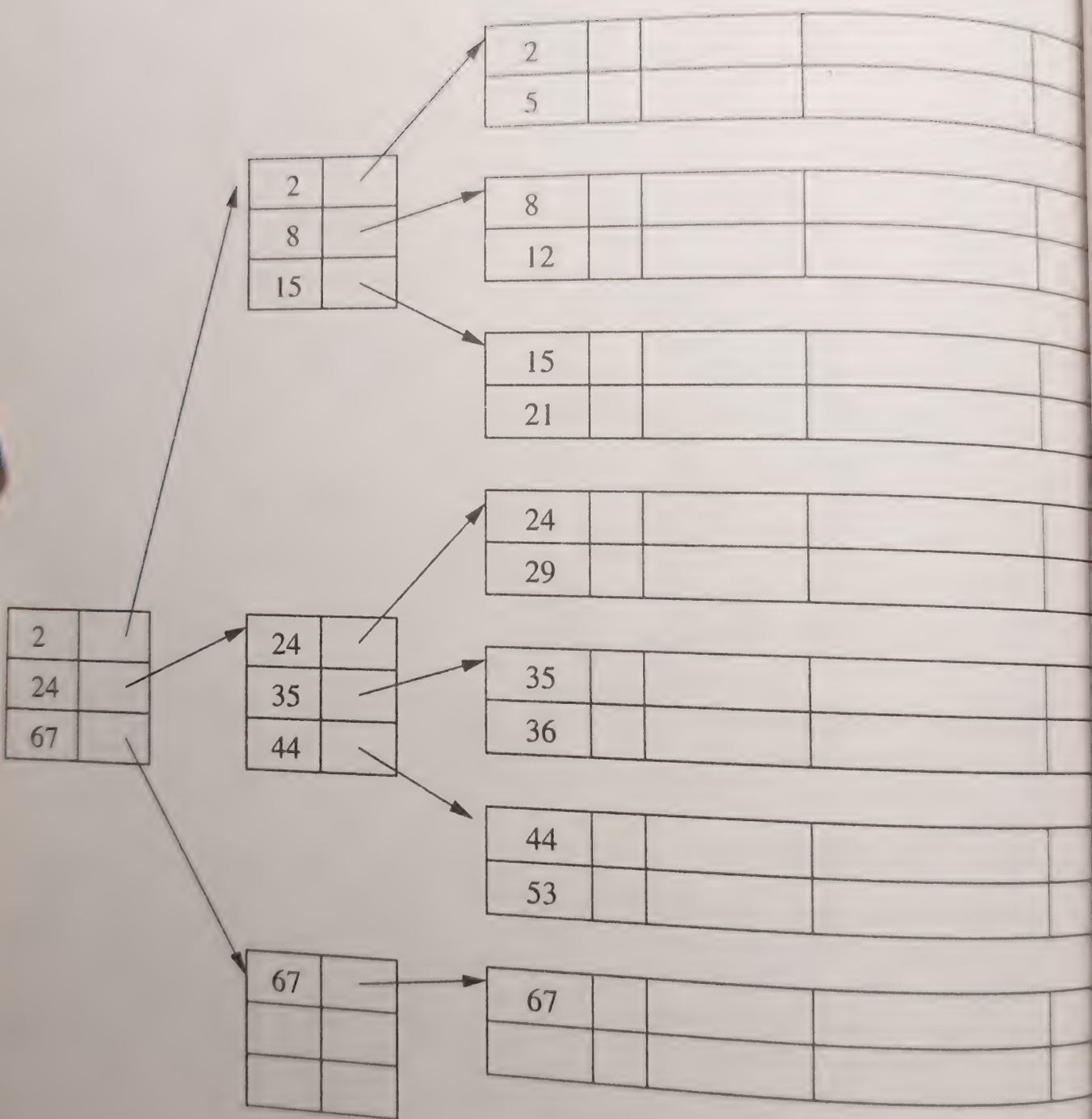



Figura 6.19: Index organizat pe 3 nivele

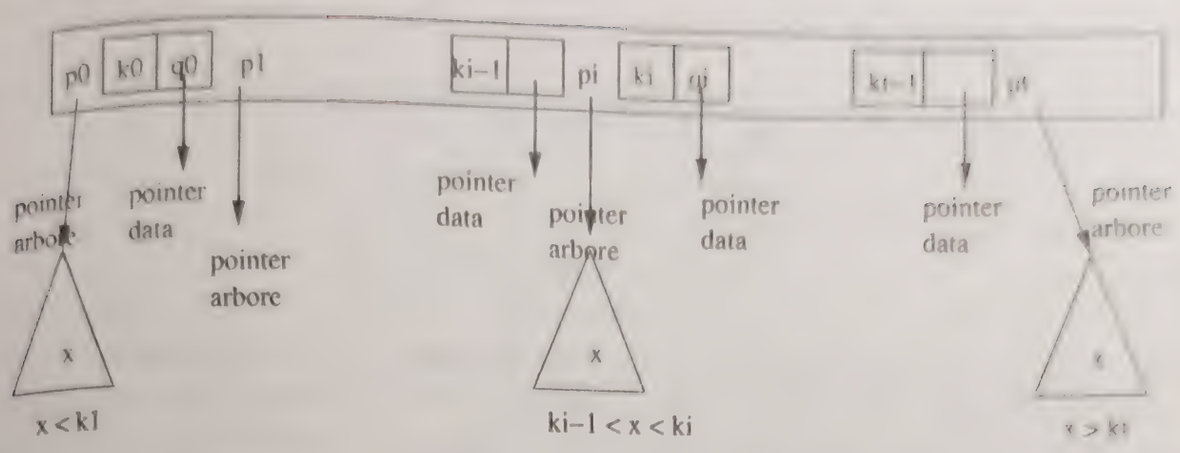



Figura 6.20: Structura unui nod al unui B-arbore

5. *fiecare nod intern are cel puțin un număr de $f-1$ chei ($f$ fii), $f$ = factorul de minimizare;*
6. *doar rădăcina poate avea mai puțin de $f-1$ chei ($f$ fii);*
7. *fiecare nod intern are cel mult $2f-1$ chei ($f$ fii);*
8. *lungimea oricărui drum de la rădăcină la o frunză este aceeași.*

Dacă fiecare nod necesită accesarea discului, atunci B-arborii vor necesita un număr minim de astfel de accesări. Factorul de minimizare va fi ales astfel încât dimensiunea unui nod să corespundă unui multiplu de blocuri ale dispozitivului de memorare. Această alegere optimizează accesarea discului.

Înălțimea $h$ unui B-arbore cu $n$ noduri și $f > 1$ satisface relația $h \leq \log_f \frac{n+1}{2}$.

Un nod $v$ al unui B-arbore poate fi implementat cu o structură statică formată din câmpurile `nrChei`, `tipNod`, `cheie[nrChei]`, `data[nrChei]` și `fiu[nrChei+1]`. Câmpul `tipNod` conține o valoare $tipNod \in \{frunza, interior\}$, câmpul `nrChei` conține numărul $t$ al cheilor din nodul $v$, tabloul `cheie[nrChei]` conține valorile cheilor memorate în nod $(k_0, k_1, \ldots, k_{t-1})$, tabloul `data[nrChei]` conține pointerii la structurile care conțin datele asociate nodului $v$ $((q_0, q_1, \ldots, q_{t-1})$, iar tabloul `fiu[nrChei]` conține pointerii la structurile care implementează fiii nodului $v$ $((p_0, p_1, \ldots, p_t)$ (vezi figura 6.20).

Căutarea într-un B-arbore este asemănătoare cu căutarea într-un arbore binar. Deoarece timpul de căutare depinde de adâncimea arborelui, `căutareBarbore` are timpul de execuție $O(\log_f n)$ ([CLR93]).

```
cautareBarbore(v, k)
   i ← 0
   while (i < v->nrChei and k > v->cheie[i]) do i ← i+1
   if (i < v->nrChei and k = v->cheie[i]) then return (v, i)
   if (v->tipNod = frunza)
      then return NULL
      else citesteMemmorieExterna(v->fiu[i])
           return cautareBarbore(v->fiu[i], k)
end
```

Operația `creeazaBarbore` creează un B-arbore vid cu rădăcina $t$ fără fii. Timpul de execuție este O(1).

```
creeazaBarbore(t)
   new t
   t->tipNod ← frunza
   t->nrChei ← 0
   scrieMemmorieExterna(t)
end
```

Dacă un nod $v$ este încărcat la maxim ($2f - 1$ chei), pentru a insera o cheie nouă este necesară spargerea acestuia. Prin spargere, cheia mediană a nodului $v$ este mutată în părintele $u$ al acestuia ($v$ este al $i$-lea fiu). Este creat un nou nod $w$ şi toate cheile din $v$ situate la dreapta cheii mediane sunt mutate în $w$. Cheile din $v$ situate la stânga cheii mediane rămân în $v$. Nodul nou $w$ devine fiu imediat la dreapta cheii mediane care a fost mutată în părintele $u$, iar $v$ devine fiu imediat la stânga cheii mediane care a fost mutată în părintele $u$. Spargerea transformă nodul cu $2f - 1$ chei în două noduri cu $f - 1$ chei (o cheie este mutată în părinte). Timpul de execuţie este $O(t)$ unde $t$ = constant.

```
procedure spargeNod(u, i, v)
   new nod w
   w->tipNod ← v->tipNod
   w->nrChei ← f-1
   for j ← 0 to f-2 do w->cheie[j] ← v->cheie[j+f]
   if (v->tipNod = interior)
      then for j ← 0 to f-1 do w->fiu[j] ← v->fiu[j+f]
   v->nrChei ← f-1
   for j ← u->nrChei downto i+1 do u->fiu[j+1] ← = u->fiu[j]
   u->fiu[i+1] ← w
   for j ← u->nrChei-1 downto i do u->cheie[j+1] ← u->cheie[j]
   u->cheie[i] ← v->cheie[f-1]
   u->nrChei ← u->nrChei + 1
   scrieMemmorieExterna(u)
   scrieMemmorieExterna(v)
   scrieMemmorieExterna(w)
end
```

Dacă un nod conţine mai puţin de $2f - 1$ chei, pentru a insera o cheie nouă în subarborele care are rădăcina nodul respectiv, se foloseşte subprogramul `insereaza_inNodIncomplet(v, k)` :

```
procedure insereaza_inNodIncomplet(v, k)
   i ← v->nrChei-1
   if (v->tipNod = frunza)
      then while (i >= 0 and k < v->cheie[i]) do
              v->cheie[i+1] ← v->cheie[i]
              i ← i-1
           v->cheie[i+1] ← k
           v-nrChei ← v->nrChei + 1
           scrieMemmorieExterna(v)
      else while (i >= 0 and k < v->cheie[i]) do i ← i-1
```

```
        i ← i+1
        citesteMemmorieExterna(v->fiu[i])
        if (v->fiu[i]->nrChei = 2f-1)
           then spargeNod(v, i, v->fiu[i])
                if (k > v->cheie[i]) then i ← i+1
        insereaza_inNodIncomplet(v->fiu[i], k)
end
```

Pentru a efectua o inserţie într-un B-arbore trebuie întâi găsit nodul în care urmează a se face inserţia. Pentru aceasta se aplică un algoritm similar cu `cautareBarbore`. Apoi cheia urmează a fi inserată. Dacă nodul determinat anterior conţine mai puţin de $2f-1$ chei, se face inserarea în nodul respectiv. Dacă acest nod conţine $2f-1$ chei, urmează spargerea acestuia. Procesul de spargere poate continua până în rădăcină. Pentru a evita două citiri de pe disc ale aceluiaşi nod, algoritmul sparge fiecare nod plin ($2f-1$ chei) întâlnit la parcurgea *top-down* în procesul de căutare a nodului în care urmează a se face inserarea. Timpul de spargere a unui nod este $O(f)$. Rezultă pentru inserţie complexitatea timp $O(f \log n)$.

```
procedure insereazaBarbore(t, k)
   v ← t
   if (v->nrChei = 2f-1)
      then new nod u
           t ← u;
           u->tipNod ← interior
           u->nrChei ← 0
           u->fiu[0]= v
           spargeNod(u, 0, v)
           insereaza_inNodIncomplet(u, k)
      else insereaza_inNodIncomplet(v, k)
end
```

Operaţia de ştergere a unei chei se desfăşoară astfel:

1. dacă nodul-gazdă al cheii ce urmează a fi ştearsă nu este frunză, atunci se efectuează o interschimbare între aceasta şi succesorul ei în ordinea naturală (crescătoare) a cheilor. Deoarece cheia succesoare se află într-o frunză, operaţia se reduce la ştergerea unei chei dintr-o frunză.
2. se şterge intrarea corespunzătoare cheii;
3. dacă nodul curent conţine cel puţin $f-1$ chei, operaţia de ştergere se consideră terminată;
4. dacă nodul curent conţine mai puţin decât $f-1$ chei, se consideră fraţii vecini;
5. dacă unul din fraţii vecin are mai mult decât $f-1$ chei, atunci se redistribuie una dintre intrările acestui frate în nodul-părinte şi una din intrarile din nodul părinte se redistribuie nodului curent (deficitar);
6. dacă ambii fraţii au exact $f-1$ chei, atunci se uneşte nodul curent cu unul dintre fraţii vecini şi cu o intrare din părinte;
7. dacă nodul-părinte devine deficitar (conţine mai puţin decât $f-1$ chei), acesta devine nod curent şi se reiau paşii 5-6.

### 6.1.5 Exerciţii

**Exerciţiul 6.1.1.** Într-un arbore AVL vid se inserează în ordine cheile $1, 2, \ldots, 2^k - 1$. Să se arate că arborele rezultat este perfect echilibrat.

**Exerciţiul 6.1.2.** Câţi biţi per nod sunt necesari pentru memorarea înălţimii nodului pentru un arbore AVL cu $n$ noduri? Care este cel mai mic arbore AVL care depăşeşte un contor de opt biţi pentru memorarea înălţimii?

**Exerciţiul 6.1.3.** Să se scrie un subprogram care generează un arbore AVL cu număr minim de noduri şi de înălţime dată $h$. Care este timpul de execuţie al programului?

**Exerciţiul 6.1.4.** Să se scrie subprograme complete care să realizeze operaţiile de inserare şi de ştergere într-un 2-3 arbore.

**Exerciţiul 6.1.5.** [HSAF93] Să se definească o alternativă orientată pe frontieră la definiţia 2-3 arborilor. Fiecare nod de pe frontieră conţine exact un element, iar semnificaţiile valorilor din nodurile interne sunt: $v\uparrow.valstg$ este cea mai mare valoare memorată într-un nod de pe frontiera subarborelui din stânga şi $v\uparrow.valdrp$ este cea mai mare valoare memorată într-un nod de pe frontiera subarborelui din mijloc. De asemenea, toate nodurile de pe frontieră se vor afla pe acelaşi nivel.

1. Să se definească o structură de date care să reprezinte un 2-3 arbore orientat pe frontieră.
2. Să se scrie subprograme care să realizeze operaţiile de căutare, inserare şi ştergere într-un 2-3 arbore reprezentat în acest fel.
3. Să se arate că noua definiţie păstrează proprietatea de clasă $O(\log n)$-stabilă.

**Exerciţiul 6.1.6.** Să se scrie programe care să construiască reprezentarea orientată pe frontieră a unui 2-3 arbore pornind de la reprezentarea pe noduri interne şi, reciproc, să construiască reprezentarea orientată pe noduri interne a unui 2-3 arbore pornind de la reprezentarea pe frontieră.

## 6.2 Dispersia (hashing)

Dispersia este o tehnică aplicată în implementarea tabelelor de simboluri. O *tabelă de simboluri* este un tip de date abstract în care obiectele sunt perechi (*nume, atribute*) şi asupra cărora se pot executa următoarele operaţii: căutarea unui nume în tabelă, regăsirea atributelor unui nume, modificarea atributelor unui nume, inserarea unui nou nume şi a atributelor sale şi ştergerea unui nume şi a atributelor sale. Exemple tipice pentru tabelele de simboluri sunt dicţionarele de sinonime (tezaurele) (nume = cuvânt şi atribute = sinonime) şi tabela de simboluri a unui compilator (nume = identificator şi atribute = (valoare iniţială, lista liniilor în care apare etc.)).

Întrucât operaţia de căutare se realizează după nume, notăm cu $\mathbb{U}$ mulţimea tuturor numelor. Implementarea tabelelor de simboluri prin tehnica dispersiei presupune:

- un tablou $(\mathtt{T}[\mathtt{i}] \mid 0 \leq i \leq p-1)$, numit şi *tabelă de dispersie (hash)*, pentru memorarea numelor şi a referinţelor la mulţimile de atribute corespunzătoare;

- o funcţie $h : \mathbb{U} \to [0, p-1]$, numită şi *funcţie de dispersie (hash)*, care asociază unui nume o adresă în tabelă.

O submulţime $S \subseteq \mathbb{U}$ este reprezentată în modul următor: pentru fiecare $x \in S$ se determină $i = h(x)$ şi numele $x$ va fi memorat în componenta $T[i]$. Valoarea funcţiei de dispersie se mai numeşte şi *index* sau *adresă în tabelă*. Dacă pentru două elemente diferite $x$ şi $y$ avem $h(x) = h(y)$, atunci spunem că între cele două elemente există *coliziune*. Pentru ca operaţia de dispersie să fie eficientă, trebuie rezolvate corect următoarele două probleme: alegerea funcţiei de dispersie şi rezolvarea coliziunilor.

### 6.2.1 Alegerea funcţiei de dispersie

Prezentăm sumar câteva tehnici elementare de alegere a funcţiei de dispersie.

**Trunchierea.** Se ignoră o parte din reprezentarea numelui. De exemplu, dacă numele sunt reprezentate prin secvenţe de cifre şi $p = 1000$, atunci $f(x)$ poate fi numărul format din ultimele trei cifre ale reprezentării: $f(62539194) = 194$.

**„Folding".** Reprezentarea numelui este partiţionată în câteva părţi şi apoi aceste părţi sunt combinate pentru a obţine indexul. De exemplu, reprezentatarea 62539194 este partiţionată în părţile 625, 391 şi 94. Combinarea părţilor ar putea consta, de exemplu, în adunarea lor: $625 + 391 + 94 = 1110$. În continuare se poate aplica şi trunchierea: $1110 \mapsto 110$. Deci $f(62539194) = 110$.

**Aritmetică modulară.** Se converteşte reprezentarea numelui într-un număr şi se ia ca rezultat restul împărţirii la $p$. Este de preferat ca $p$ să fie prim, pentru a avea o repartizare cât mai uniformă a elementelor în tabelă.

**Exemplu.** Presupunem că un nume este reprezentat printr-un şir de caractere ASCII. Notăm cu `int(`$c$`)` funcţia care întoarce codul zecimal al caracterului $c$.

```
function hash(x)
    h ← 0
    for i ← 0 to lung(x)-1 do
        h ← h + int(x[i])
    return h%p
end
```

sfex

**Multiplicare.** Fie $w$ cel mai mare număr ce poate fi memorat într-un cuvânt al calculatorului (de exemplu, $w = 2^{32}$, dacă cuvântul are 32 de biţi). Funcţia de dispersie prin multiplicare este dată de

$$h(x) = \left\lfloor p \cdot \left\{ \frac{A}{w} \right\} \right\rfloor$$

unde $\{y\} = y - \lfloor y \rfloor$ şi $A$ este o constantă convenabil aleasă. De obicei, se ia $A$ prim cu $w$ şi $p$ putere a lui 2.

### 6.2.2 Coliziunea

Există două tehnici de bază pentru rezolvarea coliziunii: înlănţuirea şi adresarea deschisă.

**Dispersie cu înlănţuire.** Numele cu aceeaşi adresă sunt memorate într-o listă liniară simplu înlănţuită (figura 6.21). Notăm cu `s.nume` câmpul care memorează numele unui simbol `s` şi cu `s.latr` câmpul ce memorează lista (sau adresa listei) de atribute a lui `s`.

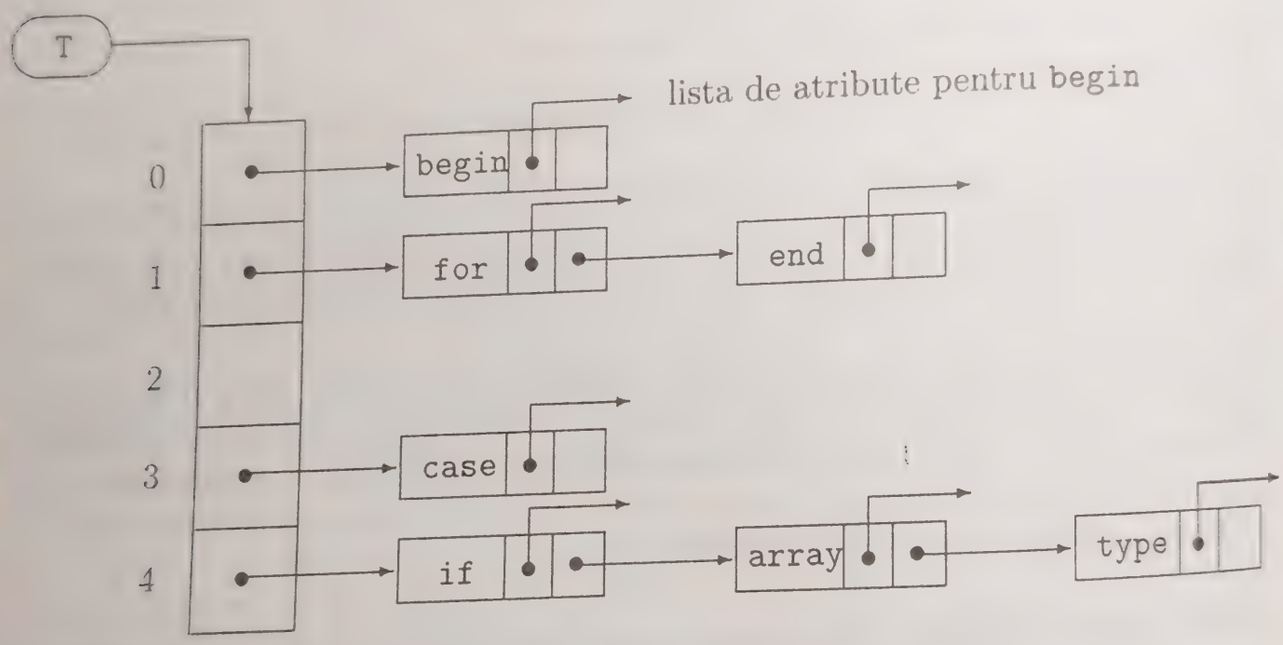



Figura 6.21: Dispersie cu înlănţuire

Căutarea unui nume în tabelă se realizează în două etape: mai întâi se calculează adresa în tabelă corespunzătoare numelui şi apoi se caută secvenţial în lista simplu înlănţuită.

```
function poz(x, T)
   i ← hash(x)
   p ← T[i]
   while (p ≠ NULL) do
      if (p->nume = x)
         then return p
         else p ← p->succ
   return NULL
end
```

Regăsirea atributelor unui nume presupune mai întâi căutarea numelui în tabelă

```
function atribute(x, T)
   p ← poz(x, T)
   if (p ≠ NULL)
```

```
        then return p->latr
        else return NULL
end
```

Operaţia de adăugare a unui element $x$ în tabelă presupune calculul lui $i = h(x)$ şi inserarea acestuia în lista $T[i]$. Pentru ca operaţia de adăugare să se realizeze cu cât mai puţine operaţii, inserarea se va face la începutul listei.

```
procedure insereaza(x, xatr, T)
   i ← hash(x)
   new(p)
   p->nume ← x
   p->latr ← xatr
   p->succ ← T[i]
   T[i] ← p
end
```

Operaţia de ştergere a numelui $x$ presupune căutarea pentru $x$ în lista $T[h(x)]$ şi eliminarea nodului corespunzător. Odată cu eliminarea numelui sunt eliminate şi atributele acestuia.

```
procedure elimina(x, T)
   i ← hash(x)
   p ← T[i]
   predp ← NULL
   while ((p ≠ NULL) and (p->nume ≠ x) do
      predp ← p
      p ← p->succ
   if (p ≠ NULL)
      then delete(p->latr) /* elimina toata lista */
           if (p = T[i])
              then T[i] ← p->succ
              else predp->succ ← p->succ
   delete(p)
end
```

**Exerciţiul 6.2.1.** Să se scrie subprograme care să realizeze modificări ale atributelor corespunzătoare unui nume. Aceste modificări vor include adăugarea de noi atribute, ştergerea de atribute sau înlocuirea de atribute.

**Teorema 6.6.** *Presupunem că tabela $T$ conţine elementele mulţimii $S \subset \mathbb{U}$. Operaţiile de adăugare şi de ştergere au complexitatea timp pentru cazul cel mai nefavorabil egală cu $O(\#S)$.*

Pentru a calcula complexitatea medie, presupunem că funcţia de dispersie distribuie uniform elementele din $\mathbb{U}$ şi că numele apar cu aceeaşi probabilitate ca argumente ale operaţiilor de căutare.

**Teorema 6.7.** *Numărul mediu de comparaţii pentru o căutare cu succes este aproximativ $1 + \frac{\beta}{2}$, unde $\beta = \frac{\#S}{p}$ este factorul de încărcare al tabelei.*

*Demonstrație.* *Metoda 1.* Știm că o căutare cu succes într-o listă liniară de lungime $m$ necesită în medie $\frac{m+1}{2}$. Lungimea medie a unei liste este $\beta$, iar probabilitatea ca elementul căutat să aparțină unei liste este $\frac{1}{p}$ (se presupune o dispersie uniformă). Numărul mediu de comparații pentru o căutare cu succes este:

$$1 + \sum_{i=0}^{p-1} \frac{\beta+1}{2} \cdot \frac{1}{p} = 1 + \frac{\beta+1}{2}$$

*Metoda 2.* Presupunem pentru moment că inserările elementelor se fac la sfârșitul listelor. Căutarea pentru al $i$-lea element inserat necesită un număr de comparații egal cu 1 plus lungimea listei la sfârșitul căreia a fost adăugat. La momentul inserării elementului $i$, lungimea medie a listelor este $\frac{i-1}{p}$. Rezultă că numărul mediu de comparații pentru o căutare cu succes este:

$$\frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{i-1}{p}\right) = 1 + \frac{\beta}{2} - \frac{1}{2p}$$

sfdem

unde $n = \#S$.

**Exercițiul 6.2.2.** Să se realizeze o implementare a dispersiei cu înlănțuire utilizând arbori binari de căutare în loc de liste liniare simplu înlănțuite.

**Dispersie cu adresare deschisă.** Pentru un nume $x \in \mathbb{U}$ se definește o secvență $h(x,i)$, $i = 0, 1, 2 \ldots$, de poziții în tabelă. Această secvență, numită și secvență de *încercări*, va fi cercetată ori de câte ori apare o coliziune. De exemplu, pentru operația de adăugare, se caută cel mai mic $i$ pentru care locația de la adresa $h(x,i)$ este liberă. O metodă uzuală de definire a secvenței $h(x,i)$ constă dintr-o combinație liniară:

$$h(x,i) = (h_1(x) + c \cdot i) \bmod p$$

unde $h_1(x)$ este o funcție de dispersie, iar $c$ o constantă. Se pot considera și forme pătratice:

$$h(x,i) = (h_1(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod p$$

O altă secvență de încercări des utilizată este următoarea:

$$h(x), h(x)-1, \ldots, 0, p-1, p-2, \ldots, h(x)+1$$

unde $h$ este o funcție de dispersie.

**Teorema 6.8.** *Pentru dispersia cu adresare deschisă, numărul mediu de încercări pentru o căutare fără succes este cel mult* $\frac{1}{1-\beta}$ $(\beta < 1)$.

*Demonstrație.* Vom presupune că $h(x,0), h(x,1), h(x,2), \ldots, h(x,p-1)$ realizează o permutare a mulțimii $\{0, 1, 2, \ldots, p-1\}$. Aceasta asigură faptul că o inserare se realizează cu succes ori de câte ori există o poziție liberă în tabelă. Notăm cu $X$ variabila aleatoare care întoarce numărul de încercări în poziții ocupate și cu $p_i$

probabilitatea $Pr(X = i)$. Evident, dacă $i > n$ ($n = \#S$), atunci $p_i = 0$. Numărul mediu de încercări $M$ în poziţii ocupate este:

$$1 + \sum_{i=0}^{n} i \cdot p_i =$$
$$1 + \sum_{i=0}^{\infty} i \cdot p_i =$$
$$1 + \sum_{i=0}^{\infty} i \cdot Pr(X = i) =$$
$$1 + \sum_{i=0}^{\infty} i \cdot (Pr(X \geq i) - Pr(X \geq i+1)) =$$
$$1 + \sum_{i=1}^{\infty} Pr(X \geq i) =$$
$$1 + \sum_{i=1}^{\infty} q_i$$

Avem $q_1 = \frac{n}{p}$, $q_2 = \frac{n}{p} \cdot \frac{n-1}{p-1} \leq \left(\frac{n}{p}\right)^2, \ldots$ de unde rezultă

$$M = 1 + \sum_{i=1}^{\infty} \beta^i = \frac{1}{1-\beta}$$

sfdem

**Corolar 6.1.** *[Knu76] Pentru dispersia cu adresare deschisă liniară, numărul mediu de încercări pentru o căutare cu succes este* $\frac{1}{2}\left[1 + \frac{1}{1-\beta}\right]$.

### 6.2.3 Exerciţii

**Exerciţiul 6.2.3.** [HSAF93] Să se proiecteze o reprezentare a unei tabele de simboluri care să permită căutarea, inserarea şi ştergerea unui nume întreg $x$ în timpul $O(1)$. Se presupune $0 \leq x < m$ şi că sunt disponibile $n + m$ registre de memorie. unde $n$ este numărul de inserări.

*Indicaţie.* Se vor utiliza două tablouri (a[i] | $0 \leq i < n$) şi (b[j] | $0 \leq j < m$) cu semnificaţiile: dacă $x$ este cel de-al $i$-lea nume inserat, atunci $a[i] = x$ şi $b[x] = i$. Se va evita iniţializarea tablourilor, deoarece aceasta necesită timpul $O(n + m)$.

**Exerciţiul 6.2.4.** [HSAF93] Fie $S = \{x_0, \ldots, x_{n-1}\}$ şi $T = \{y_0, \ldots, y_{r-1}\}$. Se presupune $0 \leq x_i < m$, $0 \leq i < n$, şi $0 \leq y_j \leq m$, $0 \leq j < r$. Utilizând ideea de la exerciţiul 6.2.3 să se scrie un algoritm care să determine dacă $S \subseteq T$. Complexitatea timp a algoritmului va fi $O(n + r)$. Deoarece $S = T$ dacă şi numai dacă $S \subseteq T$ şi $T \subseteq S$, rezultă că se poate testa dacă două mulţimi sunt echivalente în timp liniar. Care este complexitatea spaţiu a algoritmului?

**Exerciţiul 6.2.5.** [Knu76] Pentru scrierea unui compilator FORTRAN se utilizează o tabelă de simboluri pentru memorarea numelor de variabile care apar în programul FORTRAN ce se compilează. Aceste nume pot avea cel mult 8 caractere. S-a luat decizia ca dimensiunea tabelei să fie 128 şi funcţia de dispersie să fie $h(x) =$„caracterul cel mai semnificativ al numelui $x$". Este o idee bună? Justificaţi răspunsul.

**Exerciţiul 6.2.6.** [Knu76] Fie $h(x)$ o funcţie de dispersie şi $q(x)$ o funcţie de argument $x$ cu proprietatea că $x$ poate a fi determinat dacă se cunosc $h(x)$ şi $q(x)$. De exemplu, în cazul împărţirii modulare avem $h(x) = x \bmod p$ şi $q(x) = \lfloor \frac{x}{p} \rfloor$, iar în cazul dispersării prin multiplicare vom lua $h(x)$ rangurile semnificative ale lui $\{\frac{A \cdot x}{w}\}$ şi $q(x)$ celelalte ranguri (când $p$ este putere a lui 2). Să se arate că dacă se utilizează înlănţuirea, atunci este suficient să se memoreze în fiecare înregistrare $q(x)$ în loc de $x$.

**Exerciţiul 6.2.7.** [CLR93] O tabelă de dispersie de dimensiune $p$ este utilizată pentru a memora $n$ elemente, $n \leq \frac{p}{2}$. Se presupune că se utilizează adresarea deschisă pentru rezolvarea coliziunilor.

1. Presupunând că dispersia este uniformă, arătaţi că, pentru $i = 0, 1, \ldots, n-1$, probabilitatea ca a $i$-a inserare să necesite mai mult de $k$ încercări este cel mult $2^{-k}$.
2. Notăm cu $X_i$ variabila aleatoare care întoarce numărul de încercări necesare la a $i$-a inserare şi prin $X$ variabila aleatoare $\max_i X_i$. Să se arate că:
   a) $Pr(X > \log n) \leq \frac{1}{n}$.
   b) $M(X) = O(\log n)$.

**Exerciţiul 6.2.8.** Un număr mare de ştergeri într-o tabelă de dispersie cu înlănţuire conduce la o tabelă aproape vidă şi deci la irosire a spaţiului de memorie. O soluţie este înjumătăţirea tabelei. În mod similar, un număr mare de inserări conduce la dublarea dimensiunii tabelei. Presupunem că tabela este dublată când numărul elementelor memorate în tabelă este de două ori mai mare decât dimensiunea tabelei. Care este criteriul pentru înjumătăţirea tabelei?

## 6.3 Arbori digitali (tries)

Până acum am studiat numai structuri de date pentru reprezentarea mulţimilor $S$, astfel încât pentru operaţiile de căutare, inserare şi ştergere să existe algoritmi eficienţi din punctul de vedere al complexităţii timp. În această subsecţiune vom studia o structură de date care, în anumite situaţii, reduce semnificativ spaţiul de memorie necesar reprezentării mulţimii $S$. Considerăm exemplul când $S$ este un dicţionar. Se poate reduce spaţiul necesar pentru memorarea dicţionarului dacă pentru cuvintele cu rădăcina comună, aceasta este reprezentată o singură dată. Definiţia arborilor digitali are ca punct de plecare această idee.

Un arbore digital este o structură de date care se bazează pe reprezentarea digitală (cu cifre) a elementelor din mulţimea univers. Denumirea de *tri* (în unele cărţi *trie*) a fost dată de E. Fredkin (CACM, 3, 1960, pp. 490-500) şi constituie o parte din expresia din limba engleză *information-retrieval*. Un arbore digital este de fapt un arbore cu rădăcină ordonat $k$-ar (fiecare vârf are cel mult $k$ succesori), unde $k$ este numărul de cifre (litere dintr-un alfabet) necesare pentru reprezentarea elementelor mulţimii univers. Se presupune că toate elementele sunt reprezentate prin secvenţe de cifre (litere) de aceeaşi lungime $m$. Astfel, mulţimea univers conţine $m^k$ elemente.

**Exemplu.** Presupunem că elementele mulţimii universale sunt codificate prin secvenţe de trei cifre din alfabetul $\{0, 1, 2\}$. Mulţimea de chei $S = \{121, 102, 211, 120, 210, 212\}$ este reprezentată prin arborele din figura 6.22.

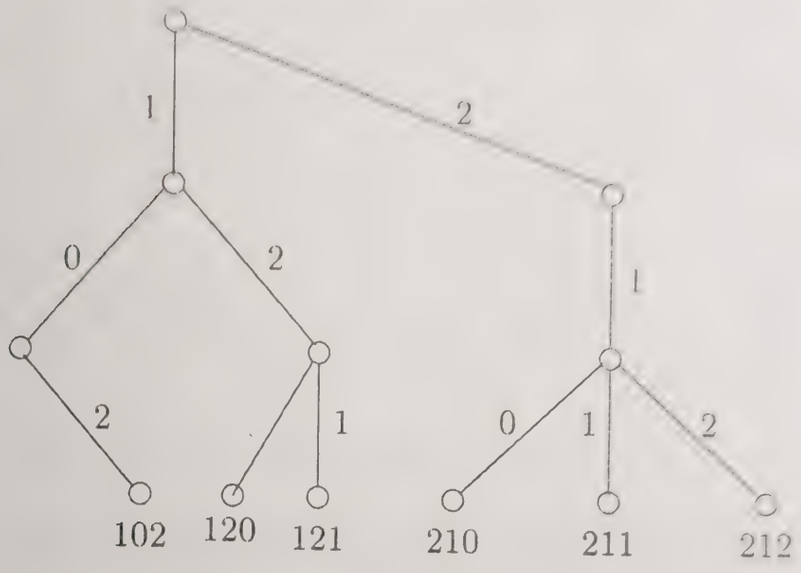



Figura 6.22: Arbore digital

Reprezentarea cuvintelor prin arbori digitali aduce o economie de memorie numai în cazul când există multe prefixe comune. În figura 6.23 este reprezentat un exemplu când spaţiul ocupat de arborele digital este mai mică decât cel ocupat de lista liniară a cuvintelor.

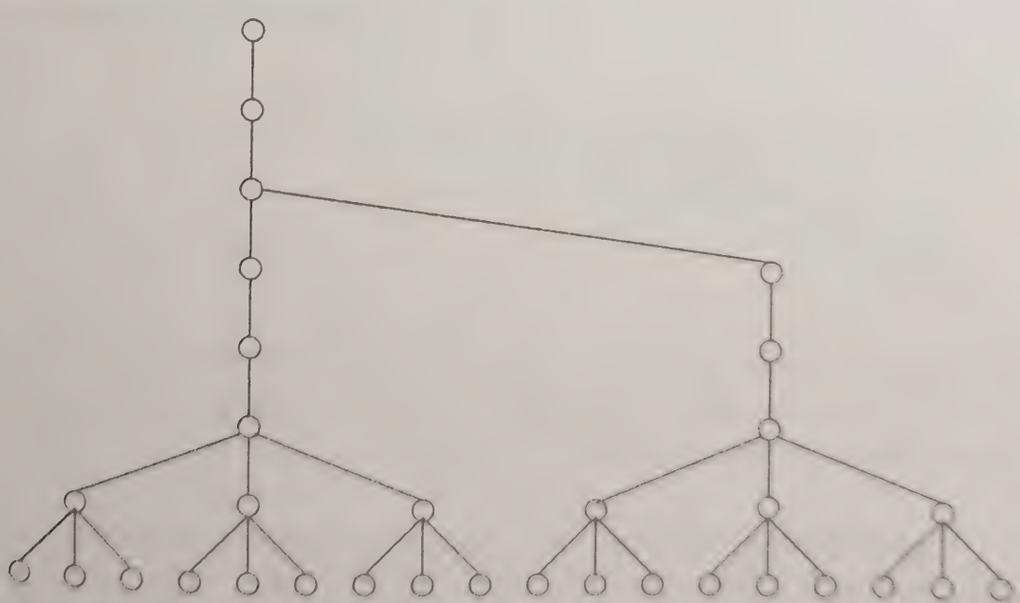


Figura 6.23: Cazul când există multe prefixe comune

Un arbore tri poate fi reprezentat în memoria calculatorului printr-o listă înlănţuită în care fiecare nod v are $k$ câmpuri de legătură, $(v.\mathtt{succ}[j] \mid 0 \leq j < k)$, ce memorează adresele fiilor lui $v$. Pentru simplitatea prezentării, presupunem că alfabetul este $\{0, \ldots, k-1\}$. Arborele din figura 6.22 este reprezentat prin structura

din figura 6.24. Elementele din mulţimea $S$ sunt gândite drept chei, iar nodurile de pe frontieră memorează informaţiile asociate acestor chei.

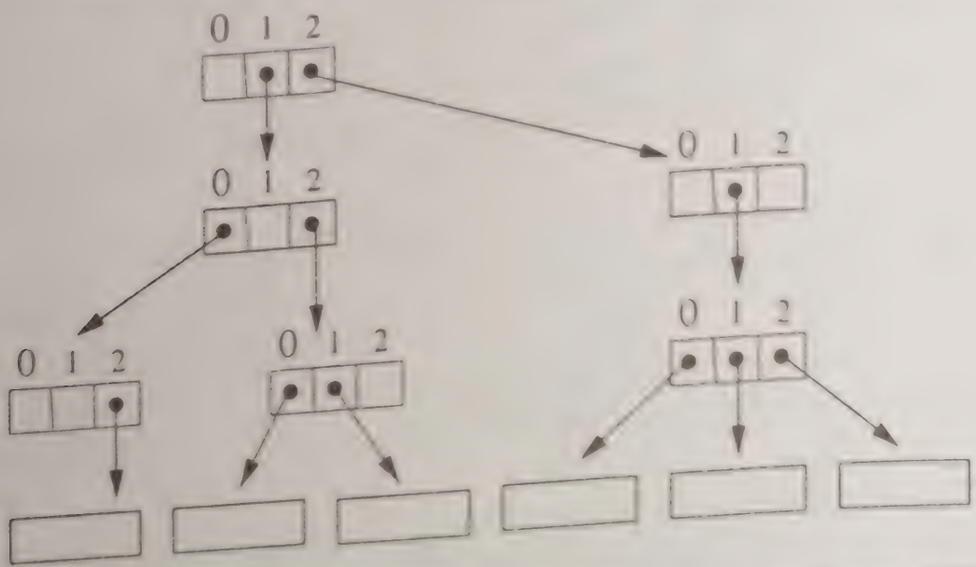



Figura 6.24: Structura de date pentru arborele din figura 6.22

Căutarea. Căutarea pentru un element $a$ în structura t constă în încercarea de a parcurge drumul în arbore descris de secvenţa $(a[0], \ldots, a[m-1])$. Parcurgerea completă a drumului înseamnă căutare cu succes ($a \in S$), iar parcurgerea parţială are semnificaţia căutării fără succes.

```
function poz(a, m, t)
begin
   i ← 0
   p ← t
   while ((p ≠ NULL) and (i < m) do
      p ← p->succ[a[i]]
      i ← i+1
   return p
end
```

Complexitatea timp pentru cazul cel mai nefavorabil este $O(m)$. De notat că aceasta nu depinde de numărul cuvintelor $n$. Pentru ca această căutare să fie mai eficientă decât căutarea binară trebuie ca $n > 2^m$.

Inserarea. Programul care realizează operaţia de inserare a unui cuvânt x la structura t simulează parcurgerea drumului descris de secvenţa $(x[0], \ldots, x[m-1])$. Pentru acele componente $x[i]$ pentru care nu există noduri în $t$ se va adăuga un nou nod ca succesor celui corespunzător lui $x[i-1]$.

Eliminarea. Considerăm mai întâi un exemplu. Dacă din structura din figura 6.22 elimină elementul reprezentat de 102, atunci este necesară şi eliminarea a două noduri (cele corespunzătoare componentelor 0 şi 2), iar ştergerea elementului 210 presupune eliminarea unui singur nod (cel corespunzător lui 0). Deci un element

care urmează a fi eliminat este împărţit în două: un prefix care este comun şi altor elemente care există în structură, şi deci nu trebuie distrus, şi un sufix care nu mai aparţine niciunui element şi deci poate fi şters. Astfel, strategia pe care o urmează programul descris aici este următoarea:

- se parcurge drumul descris de elementul x ce urmează a fi şters şi se memorează acest drum într-o stivă;
- parcurge acest drum înapoi, utilizând stiva, şi dacă pentru un nod de pe acest drum toţi succesorii sunt egali cu *nil*, atunci se elimină acel nod.

**Observaţie.** Se poate elimina utilizarea stivei, dacă la parcurgerea „dus" se determină ultimul nod care are mai mult de un succesor şi litera corespunzătoare acestui nod. Rezultă că ştergerea se poate realiza cu $O(1)$ spaţiu suplimentar. sfobs

Atât inserarea, cât şi eliminarea se realizează în timpul cel mai nefavorabil $O(n)$.

### 6.3.1 Cazul cheilor cu lungimi diferite

Există multe situaţii practice când cheile nu aceeaşi lungime. De exemplu, pentru un dicţionar de sinonime, cheile sunt cuvinte şi, evident, acestea au lungimi diferite. În plus, este foarte posibil ca un cuvânt să fie prefix al altui cuvânt. În acest caz se pune problema evidenţierii nodurilor care corespund cheilor. Soluţia cea mai la îndemână este de a adăuga un nou câmp la structura unui nod. Noul câmp va avea următoarea semnificaţie: dacă nodul corespunde unei chei, atunci câmpul va referi informaţia asociată cheii (de exemplu, lista sinonimelor); altfel, câmpul va avea valoarea NULL. Un fragment de astfel de structură este reprezentat în figura 6.25. Algoritmii de căutare, inserare şi eliminare se obţin din cei descrişi în cazul cheilor de lungimi egale prin adăugarea operaţiilor de testare şi de actualizare a câmpului nou adăugat.

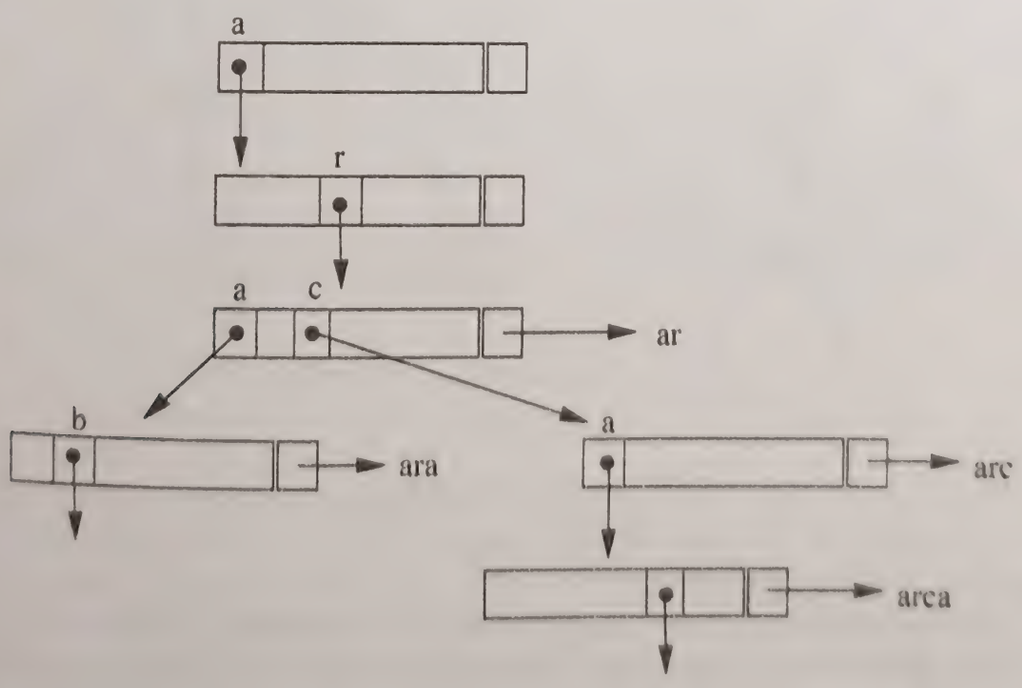



Figura 6.25: Arbore digital memorând chei cu lungimi diferite

### 6.3.2 Compactarea lanţurilor

O structură rară este o structură care memorează puţine chei. Într-o structură rară apar multe lanţuri formate din noduri intermediare cu un singur succesor. Un exemplu de structură rară este reprezentat în figura 6.26. Atât spaţiul de memorare cât şi timpul de realizare a operaţiilor pot fi îmbunătăţite prin eliminarea acestor lanţuri. Astfel, în structură vor fi păstrate numai nodurile intermediare care au cel puţin doi succesori. Aceste noduri sunt utilizate pentru a diferenţia între cheile cu un prefix comun, descris de drumul de la rădăcină la nod în arborele iniţial şi care diferă prin litera de la poziţia corespunzătoare nodului. Deoarece această poziţie nu mai este egală cu lungimea drumului de la rădăcină la nod, trebuie ca ea să fie memorată în nod. În plus, pentru că procesul de căutare a unei chei nu mai implică o parcurgere completă a cheii, există posibilitatea ca acelaşi nod de pe frontieră să fie atins cu chei diferite. Aceasta este eliminată prin memorarea cheii în nodurile de pe frontieră; atingerea nodului de pe frontieră este însoţită de testarea dacă cheia memorată coincide cu cea căutată. Aşadar, preţul plătit este un câmp nou în structura nodului, câmp care memorează poziţia din cheie testată pentru a decide succesorul pe care se continuă procesul de căutare, şi memorarea cheilor în nodurile de pe frontieră. Rezultatul compactării structurii din 6.26 este reprezentat în figura 6.27.

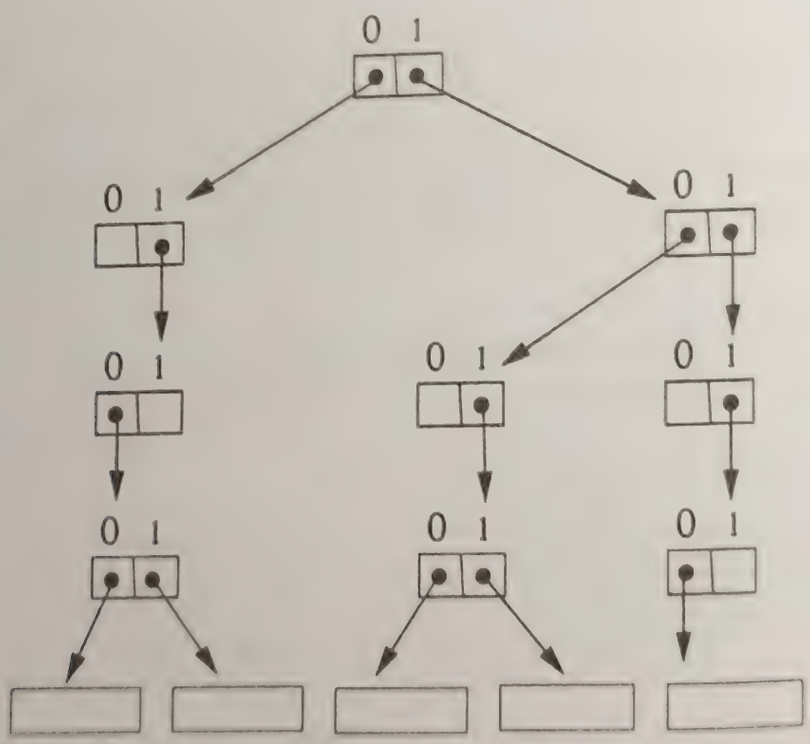



Figura 6.26: Structură rară

*Căutarea.* Operaţia de căutare este similară cazului necompactat, cu deosebirea că în fiecare nod este testată poziţia din memorată de acel nod iar atingerea unui nod de pe frontieră este însoţită de testarea cheii. De exemplu, căutarea cheii $a = 1010$ în structura din figura 6.27 se realizează astfel: Se pleacă din rădăcină. Poziţia memorată în rădăcină este 0. Deoarece $a[0] = 1$, următorul nod interogat este fiul

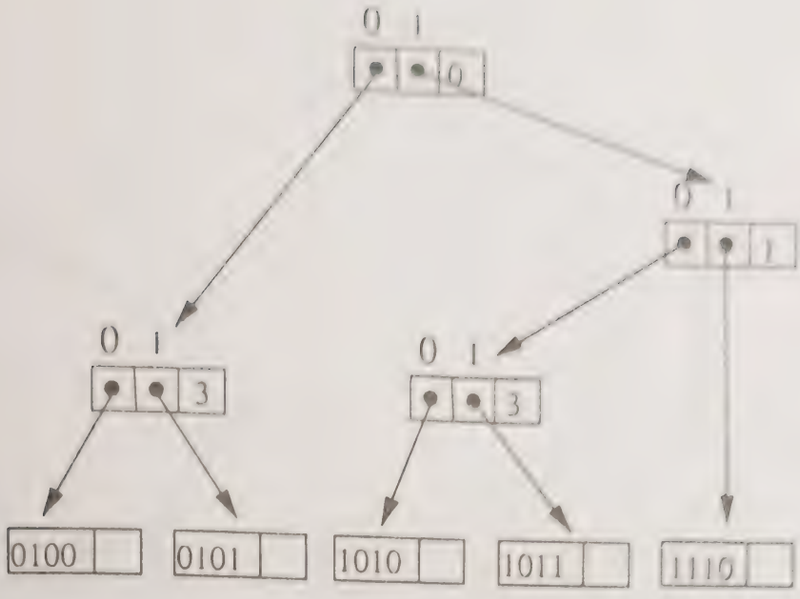



Figura 6.27: Structura din 6.26 compactată

drept. Poziţia memorată de acesta este 1 şi pentru că $a[1] = 0$, următorul nod este fiul stâng. Poziţia memorată aici este 3 şi pentru că $a[3] = 0$, următorul nod este fiul stâng. Acest nod este pe frontieră şi testăm dacă $a$ coincide cu cheia memorată în nod. Deoarece avem egalitate, căutarea se termină cu succes. De notat că acelaşi nod de pe frontieră poate fi atins şi cu cheia 1000.

Inserarea. Presupunem că dorim inserarea cheii $a = 0111$ în structura din figura 6.27. În primul pas se realizează o operaţie de căutare a acestei chei. Căutarea se termină în nodul ce memorează cheia 0101. Cele două chei diferă pe poziţia 2. Trebuie să inserăm un nou nod care să distingă după această poziţie. Deoarece $0 < 2 < 3$, acest nod va fi fiu-stânga pentru rădăcină şi nod-tată pentru nodul care distinge după poziţia 3. Acesta din urmă va fi fiu-stânga pentru nodul nou inserat. Mai adăugăm un nou nod care memorează noua cheie $a$ şi care va fi fiu-dreapta pentru nodul ce face distincţie după poziţia 2. Rezultatul este reprezentat în figura 6.28.

Eliminarea. Se realizează într-o manieră asemănătoare cazului necompactat. De exemplu, eliminarea cheii 1011 din structura din figura 6.27 se realizează astfel. Se caută nodul care memorează această cheie. În procesul de căutare se memorează drumul de la rădăcină la acest nod într-o stivă. Stiva ne ajută la parcurgerea drumului înapoi. Nodul care memorează cheia este eliminat şi se merge în nodul tată. Acest nod este eliminat şi se merge în nodul-tată, deoarece a rămas cu un singur succesor, nodul 1010. Cum acest are doi succesori, operaţia de eliminare se termină.

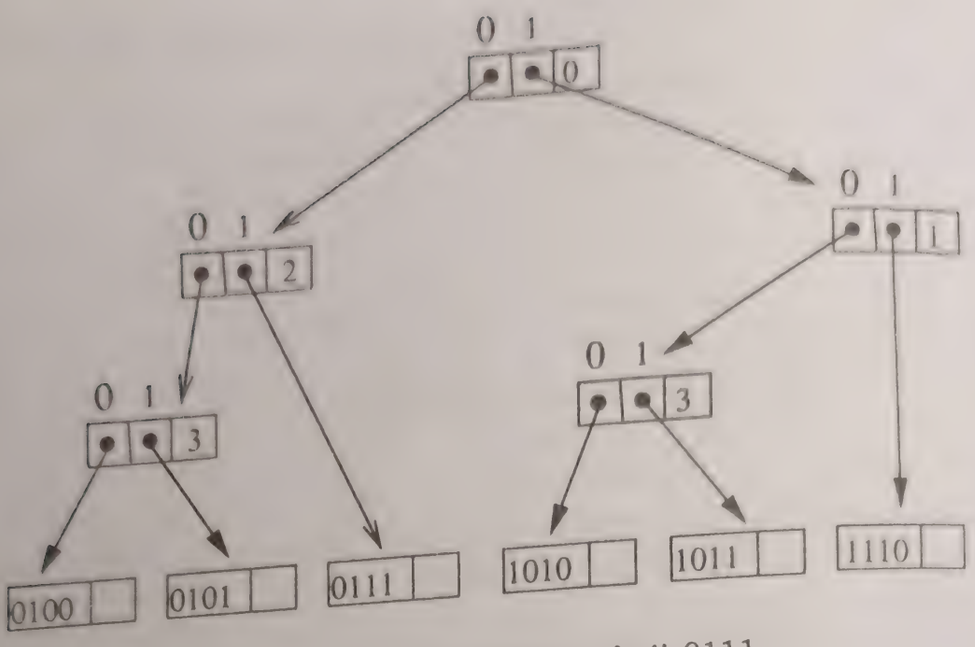



Figura 6.28: Adăugarea cheii 0111

### 6.3.3 Structuri Patricia

Un *arbore digital binar* este un arbore digital peste alfabetul $\{0, 1\}$. Exemplele din subsecţiunea 6.3.2 sunt arbori digitali binari. Într-un arbore digital binar, fiecare nod intern are exact doi fii. Aceşti arbori au următoarea proprietate: numărul nodurilor de pe frontieră este cu 1 mai mare decât numărul nodurilor interne. Dacă mai adăugăm un nod intern, atunci putem să memorăm cheile în nodurile interne, reducând astfel numărul nodurilor. Nodul pe care îl adăugă îl punem ca rădăcină. Astfel rădăcina va avea totdeauna un singur fiu, anume pe cel stâng (stabilit convenţional). Poziţia corespunzătoare rădăcinii este $-1$. Memorarea cheilor în nodurile interne se face astfel încât poziţia nodului care va memora cheia este mai mică decât sau egală cu poziţia nodului tată în arborele iniţial. După eliminarea nodurilor de pe frontieră, legăturile se refac pe principiul „legătura urmează cheia". Noua structură obţinută se numeşte *Patricia* (*Practical Algorithm to Retrieve Information Coded in Alphanumeric*). O structură Patricia corespunzătoare arborelui 6.27 este reprezentată în figura 6.30. Unul dintre marele avantaje ale acestei structuri este acela că avem un singur tip de nod. În continuare, sunt explicate operaţiile pe acest exemplu. Această structură va fi luată ca exemplu pentru explicarea operaţiilor.

**Căutarea**. Presupunem că se caută cheia $x = 1010$. Din rădăcina $a$ se coboară direct în fiul stâng $b$. Deoarece poziţia din $b$ este 0 şi $x[0] = 1$, se coboară în fiul drept $f$. Cum $x[1]$ este 0, următorul nod este $e$. Poziţia din $e$ este 3 şi pentru că $x[3] = 0$, ne deplasăm din nou în $b$. Pentru că poziţia din $b$ este 0 care este mică decât ultima poziţie testată 3, procesul de parcurgere a structurii se termină. Se compară $x$ cu cheia memorată în $b$. Deoarece avem egalitate, căutarea se termină cu succes.

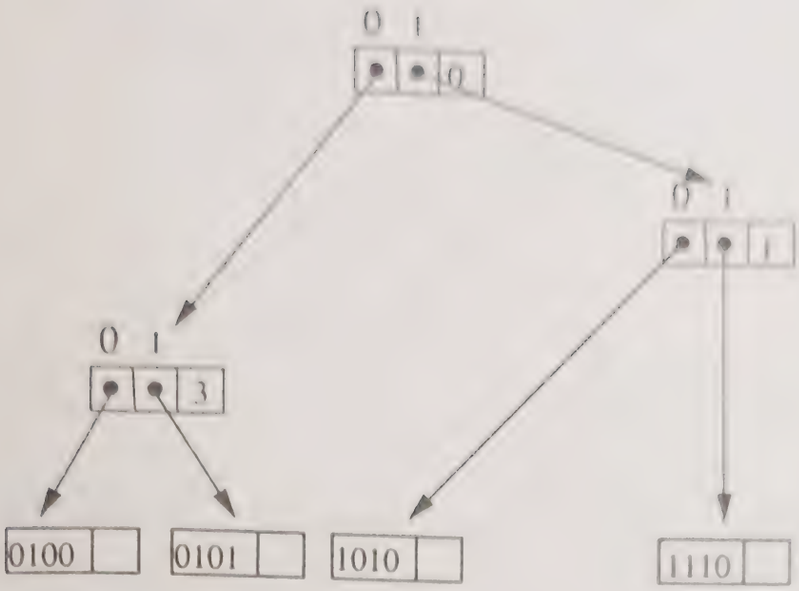



Figura 6.29: Eliminarea cheii 1011

Inserarea. Inserarea cheii $x = 1100$ presupune următorii paşi. Ca de obicei, mai întâi se caută cheia în structură. Operaţia de căutare se termină în nodul $f$. Prima poziţie pe care cheia din $f$ şi $x$ diferă este $j = 2$. Se repetă căutarea pentru cheia scurtă formată cu primele $j - 1 = 1$ caractere din $x$. Această căutare se termină în $b$ şi ultima mutare făcută pe fiul drept. Vom insera un nou nod ca fiu drept al lui $b$. Noul nod va memora cheia $x$, va face distincţie pentru poziţia $j = 2$. Deoarece $x[j] = 0$, legătura pe fiul stâng va fi la el însuşi. Legătura pe fiul drept va fi la $b$, nodul pe care s-a oprit căutarea. Rezultatul este reprezentat în figura 6.31.

**Eliminarea.** Eliminarea cheii 0111 presupune mai întâi căutarea sa. Aceasta se termină în nodul $c$. Deoarece ultima mutare a fost tot din nodul $c$ pe fiul drept, rezultă că acest nod poate fi eliminat foarte simplu. Legătura de la nodul tată $b$ este dusă direct la fiul stâng al lui $c$, $d$. Rezultatul este reprezentat în figura 6.32.

**Teorema 6.9.** *Presupunem că, pornind de la structura vidă, se crează o structură Patricia prin $n$ inserări de chei generate aleator. Atunci operaţia de căutare necesită $O(\log n)$ comparaţii în medie.*

### 6.3.4 Exerciţii

**Exerciţiul 6.3.1.** (Extinderea tipului de dată.) Fie `t` un arbore digital. Notăm cu $M(t)$ mulţimea reprezentată de acest arbore. Să se scrie următoarele subprograme:

1) `reuniune(t1,t2)` care realizează operaţia $M(t) = M(t1) \cup M(t2)$.
2) `intersectie(t1,t2)` care realizează operaţia $M(t) = M(t1) \cap M(t2)$.
3) `minus(t1,t2)` care realizează operaţia $M(T) = M(t1) \setminus M(t2)$.

**Exerciţiul 6.3.2.** Să se scrie o procedură `triInLista(t, L)` care având la intrare un arbore digital `t` construieşte lista liniară `L` ce conţine toate elementele din mulţimea reprezentată de `t`, în ordine lexicografică.

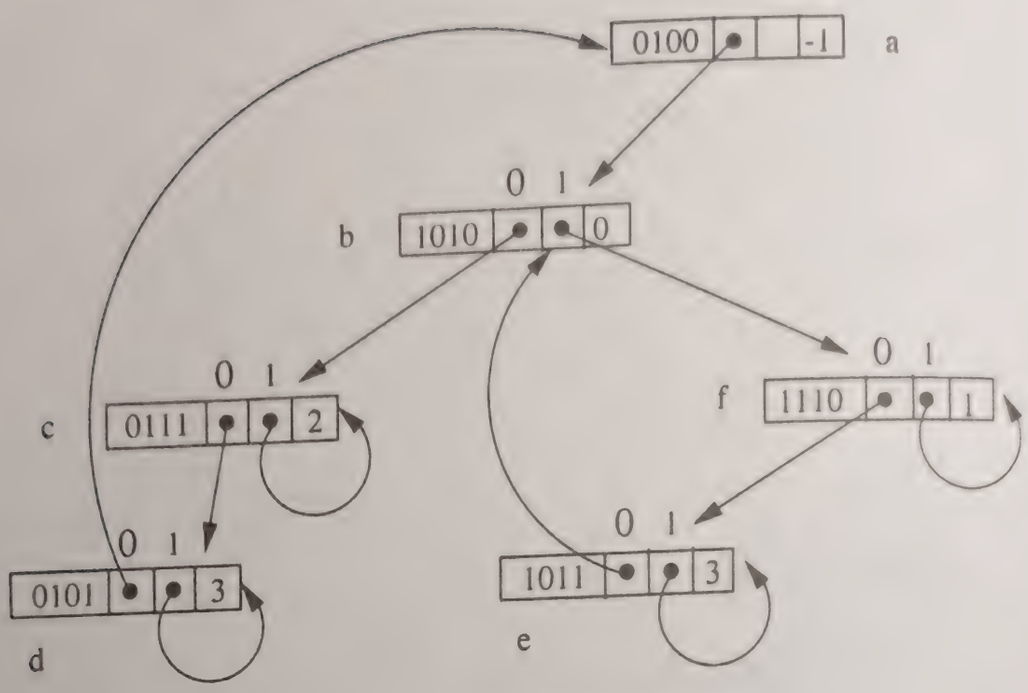



Figura 6.30: Structură Patricia

**Exerciţiul 6.3.3.** Să se descrie în limbaj algoritmic operaţiile pentru arborii digitali cu chei de lungimi diferite.

**Exerciţiul 6.3.4.** Să se descrie în limbaj algoritmic operaţiile pentru arborii digitali compactaţi.

**Exerciţiul 6.3.5.** Explicaţi modul în care se elimină cheia 1010 din structura Patricia 6.30.

**Exerciţiul 6.3.6.** Să se descrie în limbaj algoritmic operaţiile pentru structurile Patricia.

## 6.4 Referinţe bibliografice

Deşi scrisă în 1973, [Knu76] rămâne o carte de referinţă în domeniu. Multe aspecte ale căutării studiate în acest capitol sunt inspirate de aici. Completări la structurile de date studiate aici, precum şi alte structuri, mai pot fi găsite în [Mel84a, Sed88, HSAF93, Hei96, Wei92]. Pentru aspectele teoretice privind structurile de căutare recomandăm [Knu76, Mel84a, Mel79].

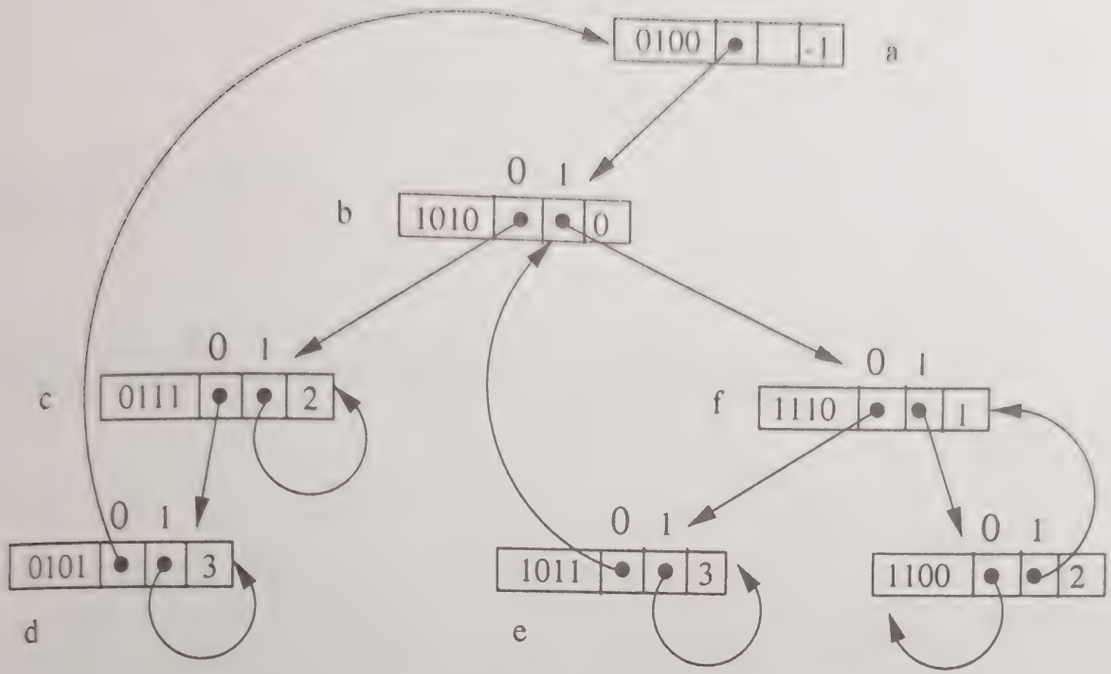



Figura 6.31: Structură Patricia: inserarea cheii 1100

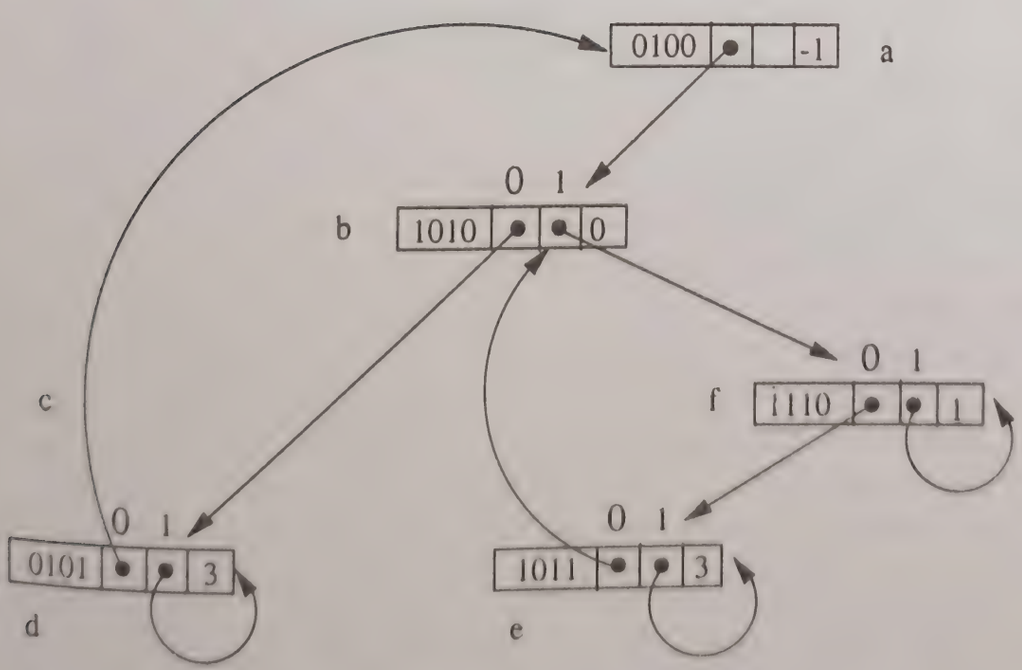



Figura 6.32: Structură Patricia: eliminarea cheii 0111

MATEMATICĂ

# Capitolul 7

# Căutare peste şiruri

Considerăm şirurile ca tipuri de dată abstracte în care:

- obiectele sunt secvenţe $x_0 \cdots x_{n-1}$ cu componentele $x_i$ aparţinând unui tip abstract ale cărui elemente le numim *caractere*;
- setul de operaţii include inserarea şi ştergerea de caractere, concatenarea a două şiruri şi regăsirea unui şir ca subşir al altuia.

Sistemele de procesare a textelor utilizează acest tip pentru editarea textelor. Un text poate fi privit ca un şir de litere, cifre, şi caractere speciale. Tipul `String` şi fişierele text constituie reprezentări ale tipului abstract şir. Un caz particular îl constituie şirurile binare, construite peste alfabetul $\{0, 1\}$. Acestea sunt utilizate la memorarea imaginilor grafice, de exemplu. În acest capitol prezentăm algoritmi pentru o singură operaţie peste şiruri şi anume regăsirea unui şir ca subşir al altuia, cunoscută şi sub numele de *căutare peste şiruri*.

Fie tipul abstract `Character` în care obiectele sunt caracterele. Fără să restrângem generalitatea, presupunem că şirurile sunt reprezentate prin tablouri unidimensionale de caractere. Problema căutării peste şiruri poate fi formulată astfel: fiind date două şiruri $s = s_0 \cdots s_{n-1}$ (subiectul sau textul) şi $p = p_0 \cdots p_{m-1}$ (patternul), să se proiecteze un algoritm eficient care să decidă dacă $p$ este subşir al lui $s$. În cazul când patternul apare de mai multe ori în text, interesează poziţia unei apariţii (de regulă, prima). De exemplu, în textul $s = aabaccabaac$ patternul $p = bbaabbcc$ nu apare niciodată , iar patternul $aba$ apare de două ori.

Algoritmii care rezolvă problema de căutării peste şiruri au o istorie interesantă [Sed88]. În 1970, S.A. Cook a demonstrat un rezultat teoretic despre un anumit tip abstract de maşină, unde se presupunea existenţa unui algoritm de căutare peste şiruri ce necesită un timp proporţional cu $n + m$ în cazul cel mai nefavorabil. D.E. Knuth şi V.R. Pratt au utilizat construcţia laborioasă din teorema lui Cook şi au elaborat un algoritm care, mai apoi, a fost rafinat într-un algoritm practic şi simplu. J.H. Morris a descoperit acelaşi algoritm în timpul implementării unui editor de texte. Este unul dintre numeroasele exemple când un rezultat pur teoretic poate conduce la rezultate cu aplicabilitate imediată. Algoritmul dat de Knuth, Morris şi Pratt a fost publicat abia în 1976. Între timp, R.S. Boyer şi J.S. Moore (şi indepedent W. Gosper) au descoperit un algoritm care este mult mai rapid în

multe situaţii. În 1980, R.M. Karp şi M.O. Rabin au proiectat un algoritm cu o descriere foarte simplă şi care poate fi extins la texte şi patterns bidimensionale deci foarte util la procesarea imaginilor grafice.

## 7.1 Căutarea naivă

Un algoritm simplu dar, după cum vom vedea, ineficient este următorul: pentru fiecare poziţie posibilă în text se testează dacă patternul se potriveşte peste subtextul care începe la acea poziţie.

```
function pmNaiv(s, n, p, m)
   i ← 0
   while (i < n-m) do
      i ← i+1
      j ← 0
      whiile (s[i+j] ≠ p[j]) do
         if (j = m-1)
            then return i /* a gasit prima aparitie a lui p*/
            else j ← j + 1
   return -1 /* p nu apare in s */
end
```

Procedura `PMNaiv` va întoarce valoarea 0, dacă patternul nu este subşir al subiectului (textului), sau poziţia $i$ de la care începe prima apariţie a patternului în subiect (text). Complexitatea timp a algoritmului $O(n \cdot m)$.

## 7.2 Algoritmul Knuth-Morris-Pratt

Ideea algoritmului este următoarea: Atunci când s-a întâlnit o nepotrivire, i.e. caracterul curent $p_j$ din pattern este diferit de caracterul curent $s_i$ din text, caracterele $s_{i-1}, \ldots, s_{i-j+1}$ din text sunt cunoscute deoarece ele există în pattern. Poziţia de start pentru următoarea comparaţie dintre pattern şi text poate fi determinată exploatând această informaţie. Ca urmare, întoarcerile în text sunt eliminate. Astfel că algoritmul va consta din doua etape:

1. Se proceseaza mai întâi patternul în $O(m)$ şi se creează o structură de date.
2. Folosind structura de date creată la prima etapă, se procesează textul fără a mai executa întoarceri.

Există două moduri de a preciza o apariţie a unui pattern în text:

- prin poziţia de start: spunem că $p$ se potriveşte în $s$ la poziţia de start $i$;
- prin poziţia de sfârşit: spunem că $p$ se potriveşte în $s$ la poziţia de sfârşit $i$

O nepotrivire o reprezentăm grafic ca în figura 7.1. Avem $s_{i-j} = p_0, \ldots, s_{i-1} = p_{j-1}, s_i \neq p_j$. Vom utiliza această informaţie pentru a determina cel mai lung prefix propriu al lui $p$ care se potriveşte în $s$ la poziţia de sfârşit $i$. Notăm acest prefix cu $F[i,j]$. Determinarea funcţiei $F[i,j]$ se face într-o manieră recursivă. Pentru moment presupunem $p = p_0 \cdots p_j$.

1. $F[i,0]$ este şirul vid (de lungime zero).

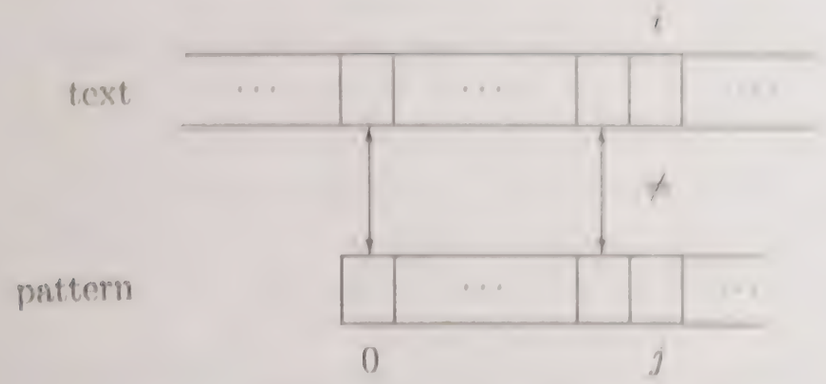



Figura 7.1: Reprezentarea grafică a nepotrivirii $s_i \neq p_j$

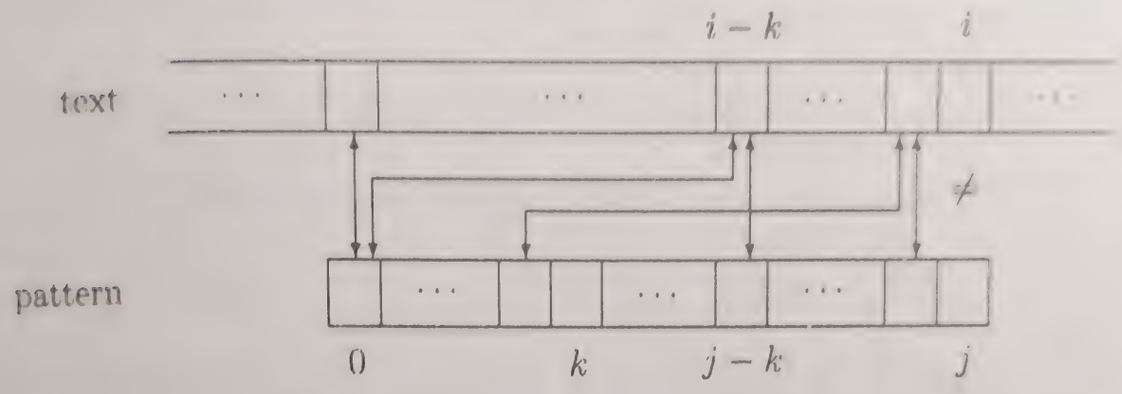



Figura 7.2: $s_i \neq p_k \Rightarrow F[i,j] = F[i,k]$

2. Presupunem $j > 0$. Se consideră cel mai mare prefix $q$ al lui $p$ care se potriveşte în $s$ la poziţia de sfârşit $i-1$. Presupunem $q = p_0 \cdots p_{k-1}$ (figura 7.2) şi notăm $p' = qp_k$. Dacă $p_k = s_i$ atunci $F[i,j] = p'$. Altfel, $p'$ se află în aceeaşi situaţie ca $p$ şi deci avem $F[i,j] = F[i,k]$.

Presupunem că pentru patternul $p$ s-a determinat $F[i,j]$ pentru orice prefix $p_0 \cdots p_j$ al lui $p$. În momentul apariţiei unei nepotriviri $p_j \neq s_i$, $F[i,j] = p_0 \cdots p_k$ este cel mai mare prefix al lui $p_0 \cdots p_j$ care se potriveşte în $s$ la poziţia de sfârşit $i$ şi deci se poate continua cu comparaţia dintre $s_{i+1}$ şi $p_{k+1}$.

Convenim să notăm $F[i,j] = k$ în loc de $F[i,j] = p_0 \cdots p_k$ (orice prefix este identificat în mod unic de poziţia sa de sfârşit în pattern).

Cum poate fi calculată eficient funcţia $F[i,j]$? Notăm cu $f[j]$ valoarea $k$ determinată în pasul inductiv din definiţia lui $F[i,j]$. Cunoscând $f$, $F[i,j]$ poate fi calculată foarte simplu:

```
k ← f[j]
while (k ≠ 0) and (p[k] ≠ s[i]) do
   k ← f[k]
F[i,j] ← k
```

Funcţia $f$ poate fi calculată explorând numai patternul: dacă $f[j] = k$, atunci $p_0 \cdots p_{k-1}$ este de fapt cel mai lung prefix al lui $p$ care se potriveşte în $p$ la poziţia de sfârşit $j-1$. Ca şi $F$, funcţia $f$ poate fi definită recursiv:

- Dacă $j = 0$, atunci $f[0] = -1$ (putem presupune că $p_0 \cdots p_{-1}$ desemnează şirul vid);
- Presupunem $j > 0$. Fie $k = f[j-1]$ (care este definită). Rezultă că $p_0 \cdots p_{k-1}$ este cel mai lung prefix al lui $p_0 \cdots p_j$ care se potriveşte în $p$ la poziţia de sfârşit $j-2$. Dacă $p_k = p_{j-1}$, atunci $f[j] = k+1$. Altfel, se caută cel mai lung prefix al lui $p_0 \cdots p_k$ care se potriveşte în $p$ la poziţia de sfârşit $j-2$. Adică se face $k = f[k]$ şi se repetă raţionamentul de mai sus.

Datorită rolului pe care-l joacă în timpul căutării, funcţia $f$ se numeşte *funcţie eşec* (*failure function*). Subprogramul Pascal care calculează $f$ are o structură foarte simplă:

```
procedure Determina_f (p, m, f)
   f[0] ← -1
   for j ← 1 to m-1 do
       k ← f[j-1]
       while ((k ≠ -1) and (p[j-1] ≠ p[k])) do
          k ← f[k]
       f[j] ← k+1
end
```

Notăm cu KMP (de la iniţialele celor trei autori) subprogramul care realizează efectiv operaţia de pattern-matching:

```
function KMP(s, n, p, m, f)
   i ← 0
   j ← 0
   while (i < n) do
      while (j ≠ -1) and (p[j] ≠ s[i]) do
         j ← f[j]
      if (j = m-1)
         then return i-m+1 /* gasit p in s */
         else i ← i+1
              j ← j+1
   return -1 /* p nu apare in s */
end
```

Funcţia failure pentru patternul *abaabaaabc* este reprezentată în figura 7.3. Privind mai atent această funcţie se observă că algoritmul poate fi îmbunătăţit. De exemplu, dacă apare o nepotrivire între un caracter $s_i$ din text şi al cincilea caracter din pattern, atunci algoritmul continuă cu comparaţia dintre $s_i$ şi al doilea caracter din pattern. Dar noi ştim că între acestea există nepotrivire pentru că al doilea şi al cincilea caracter din pattern sunt identice.

**Teorema 7.1.** *Algoritmul* KMP *execută totdeauna cel mult* $n + m$ *comparaţii de caractere.*

Deşi algoritmul KMP execută mai puţine comparaţii decât cel naiv, în practică KMP nu se dovedeşte a fi mai rapid în mod semnificativ pentru că sunt foarte rare situaţiile când patternul are o structură repetitivă.

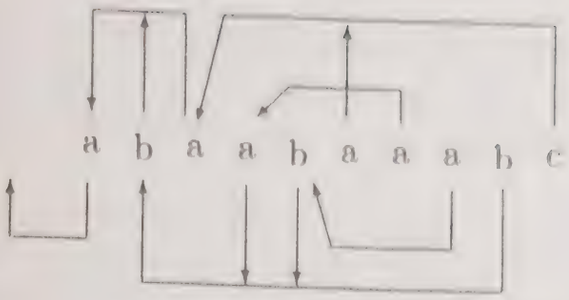



Figura 7.3: Funcţia eşec pentru algoritmul KMP

## 7.3 Algoritmul Boyer-Moore

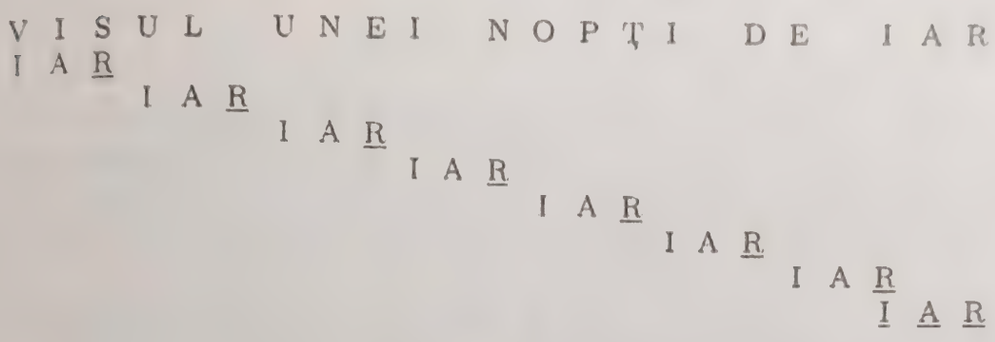



Figura 7.4: Algoritmul Boyer-Moore

Caracterele patternului sunt explorate de la dreapta la stânga. Un exemplu ce arată modul de lucru al algoritmului este reprezentat în figura 7.4. Prima dată se compară R (ultimul caracter din pattern) cu S (al treilea caracter din text). Deoarece S nu apare în pattern, se deplasează patternul cu trei poziţii (lungimea sa) la dreapta. Apoi se compară R cu caracterul spaţiu. Nici caracterul spaţiu nu apare în pattern aşa că acesta se deplasează din nou la dreapta cu trei poziţii. Procesul continuă până când R este comparat cu I (al 21-lea caracter din text). Deoarece I apare în pattern pe prima poziţie se deplasează patternul la dreapta cu două poziţii. Apoi se compară R cu R, deci există potrivire. Se continuă comparaţia cu penultimul caracter din pattern (de fapt, al doilea) şi precedentul din text (al 22-lea). Se obţine din nou potrivire şi se compară următoarele două caractere de la stânga (primul din pattern şi al 21-lea din text). Deoarece există potrivire şi patternul a fost parcurs complet, rezultă că s-a determinat prima apariţie a patternului în text.

Ca şi în cazul algoritmului `KMP`, există o etapă de preprocesare a patternului. Această preprocesare construieşte o funcţie care, pentru fiecare caracter din alfabet, precizează câte poziţii este deplasat patternul la dreapta atunci când apare o nepotrivire. Notăm cu `detSalt(p, salt)` procedura care calculează funcţia `salt`. Pentru patternul IAR avem $salt['R'] = 0$, $salt['A'] = 1$ şi $salt['I'] = 2$. În ipoteza căutării repetate pentru acelaşi pattern, este preferabil ca funcţia `salt` să fie calculată o singură dată şi apoi transmisă ca parametru algoritmului de căutare.

Descrierea algoritmului este foarte simplă:

```
function BM(s, n, p, m, salt)
    i ← m-1
    j ← m-1
    repeat
       if (s[i] = p[j])
          then i ← i-1
               j ← j-1
          else if (m-j > salt[s[i]])
                  then i ← i+m-j
                  else i ← i+salt[s[i]]
               j ← m-1
    until ((j<0) or (i>n-1))
    if (j < 0)
       then return i+1
       else return -1

end
```

În cazul unei nepotriviri de forma

$$\begin{array}{ccccc} & & \mathrm{i} & & \\ \ldots & \mathrm{R} & \mathrm{A} & \mathrm{R} & \cdots \\ & \underline{\mathrm{I}} & \mathrm{A} & \mathrm{R} & \\ & \mathrm{j} & & & \end{array}$$

valoarea lui $i$ este incrementată cu $3 - j = 3 > 0 = salt['R']$.

**Teorema 7.2** ([Sed88]) *Algoritmul* BM *execută totdeauna cel mult $m+n$ comparaţii de caractere şi aproximativ $\frac{n}{m}$ salturi când alfabetul nu este mic şi patternul nu este prea lung.*

**Observaţie.** Dacă alfabetul are numai două caractere (cazul şirurilor binare), atunci performanţele algoritmului $BM$ nu sunt cu mult mai bune decât cele ale căutării naive. În acest caz se recomandă împărţirea şirurilor în grupe cu un număr fixat de biţi. Fiecare grupă reprezintă un caracter. Dacă dimensiunea unei grupe este $k$, atunci vor exista $2^k$ caractere, i.e. dintr-un alfabet mic obţinem unul cu multe caractere. Totuşi, $k$ va trebui ales suficient de mic pentru ca dimensiunea tabelei de salturi să nu fie prea mare.

## 7.4 Algoritmul Rabin-Karp

Acest algoritm utilizează tehnica tabelelor de dispersie (hash). Un simbol este o secvenţă de $m$ caractere. Să ne imaginăm că toate simbolurile posibile sunt memorate într-o tabelă de dispersie foarte mare, astfel încât nu există coliziune. A testa dacă patternul $p$ coincide cu un subşir de lungime $m$ din text este echivalent cu a testa dacă funcţia de dispersie $h$ dă aceeaşi valoare pentru ambele simboluri. Există avantajul că pentru pattern funcţia de dispersie este calculată o singură dată. Pentru ca algoritmul să fie eficient, funcţia de dispersie trebuie definită în aşa fel

încât, atunci când se face o deplasare la dreapta în text, calculul valorii funcţiei pentru următorul simbol să fie cât mai simplu. Timpul necesar efectuării acestui calcul trebuie să fie cu mult mai mic decât cel necesar comparării a două şiruri de lungime $m$.

Un mod de a defini funcţia de dispersie este următorul: Se consideră fiecare şir de $m$ caractere ca fiind reprezentarea unui număr întreg în baza $d$, unde $d$ este numărul maxim de caractere. Numărul zecimal corespunzător şirului $s[i..i+m-1]$ este:

$$x = s[i]d^{m-1} + s[i+1]d^{m-2} + \cdots + s[i+m-1]$$

Funcţia de dispersie $h$ va fi definită prin $h(s[i..i+m-1]) = x \bmod q$, unde $q$ este un număr prim foarte mare. O deplasare la dreapta în text corespunde înlocuirii lui $x$ cu:

$$(x - s[i]d^{m-1})d + s[i+m]$$

În formulele de mai sus s-a presupus caracterele $0, \ldots, d-1$. Pentru a interpreta caracterele textului drept cifre, considerăm o funcţie *index* care asociază fiecărui caracter numărul său de ordine în alfabet (similar funcţiei `Ord` din Pascal). Toate acestea conduc la următorul algoritm foarte simplu:

```
functiom RK(s, n, p, m,)
   dlam1 ← 1 /* calculeaza d la puterea m-1 */
   for i ← 1 to m-1 do
       dlam1 ← (d*dlam1) mod q
   hp ← 0 /* h(p) */
   for i ← 0 to m-1 do
       hp ← (hp*d+index(p[i])) mod q
   hs ← 0 /* h(s[1..m] */
   for i ← 0 to m-1 do
       hs ← (hs*d+index(s[i])) mod q
   i ← 0
   while ((hp ≠ hs) and (i <= n-m)) do
      if (hp = hs)
         then return i
      hs ← (hs+d*q-index(s[i])*dlam1) mod q
      hs ← (hs*d+index(s[i+m])) mod q
      i ← i+1
   return -1
end
```

Adunarea cu $d * q$ la recalculul lui `hs` se face pentru a fi siguri că se obţine un număr pozitiv. Alegerea $d = 32$ s-a făcut pentru ca operaţia de înmulţire să se efectueze foarte simplu (întregii din `Longint` sunt reprezentaţi pe 32 de biţi). Merită notat faptul că din cauza eventualelor coliziuni, testul final mai trebuie dublat de verificarea $p = s[i..i+m-1]$.

Deşi teoretic, în cazul cel mai nefavorabil, algoritmul `RK` are o complexitate timp proporţională cu $n \cdot m$, în practică s-a dovedit a executa aproximativ $n + m$ paşi.

## 7.5 Expresii regulate

În această secţiune considerăm cazul când patternul constituie doar o specificaţie a ceea ce se caută în sensul că el desemnează o mulţime de şiruri pentru care se caută. Numim o astfel de specificaţie *pattern generalizat*. Un exemplu de pattern generalizat îl constituie utilizarea caracterului *wildcard* (a se vedea şi exerciţiul 7.2). Un *wildcard* este un caracter special, să zicem „?", care se potriveşte peste orice alt caracter. De exemplu, patternul generalizat „ab?a" specifică de fapt mulţimea {abaa, abba, abca, abda, ...}. Orice subşir al textului care aparţine acestei mulţimi se va potrivi peste acest pattern. Un alt mod de a specifica patternuri generalizate îl constituie expresiile regulate.

**Definiţia 7.1.** *Mulţimea* expresiilor regulate *peste alfabetul $A$ este definită recursiv astfel:*

- *orice caracter din $A$ este o expresie regulată;*
- *dacă $e_1, e_2$ sunt expresii regulate, atunci $(e_1e_2)$, $(e_1 + e_2)$ şi $e_1^*$ sunt expresii regulate.*

*Mulţimea de şiruri (limbajul) $L(e)$ definit de o expresie regulată $e$ este definit recursiv astfel:*

- *dacă $e$ este un caracter, atunci $L(e) = \{e\}$;*
- *dacă $e = (e_1e_2)$, atunci $L(e) = L(e_1)L(e_2) = \{w_1w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;*
- *dacă $e = (e_1 + e_2)$, atunci $L(e) = L(e_1) \cup L(e_2)$;*
- *dacă $e = e_1^*$, atunci $L(e) = \cup_k L(e_1^k)$, unde $L(e_1^0) = \{\varepsilon\}$, $L(e_1^{k+1}) = L(e_1^k)L(e_1)$ ($\varepsilon$ este şirul vid (de lungime zero)),*

**Exemplu.** Fie alfabetul $A = \{a, b, c\}$. Avem $L(a(b + a)c) = \{abc, aac\}$ şi $L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.

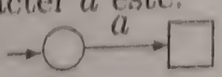
sfex

Deoarece $L(e_1(e_2e_3)) = L((e_1e_2)e_3)$, vom scrie $e_1e_2e_3$ pentru a desemna una din cele două expresii. Vom aplica astfel de convenţii de omitere a parantezelor ori de câte ori este posibil.

În continuare ne vom ocupa de modul în care expresiile regulate pot fi utilizate în procesul de căutare într-un text. Mai întâi plecăm de la observaţia că patternul şi funcţia eşec de la algoritmul KMP pot fi gândite ca reprezentând un automat. În figura 7.5 este reprezentat automatul corespunzător patternului şi funcţiei eşec din figura 7.3. Acest automat este reprezentat ca un digraf în care vârfurile reprezintă stările automatului iar arcele relaţia de tranziţie. Tranziţiile descriu comportarea automatului. De exemplu, dacă automatul din figura 7.5 se află în starea 3 şi caracterul curent din text este 'b', atunci automatul va trece în starea 4. Altfel va trece în starea 1. Dacă automatul ajunge în starea finală (desenată prin pătrat), atunci s-a determinat o apariţie a patternului în text.

Noţiunea de automat o extindem la expresii regulate după cum urmează:

1. Automatul definit de un caracter $a$ este:



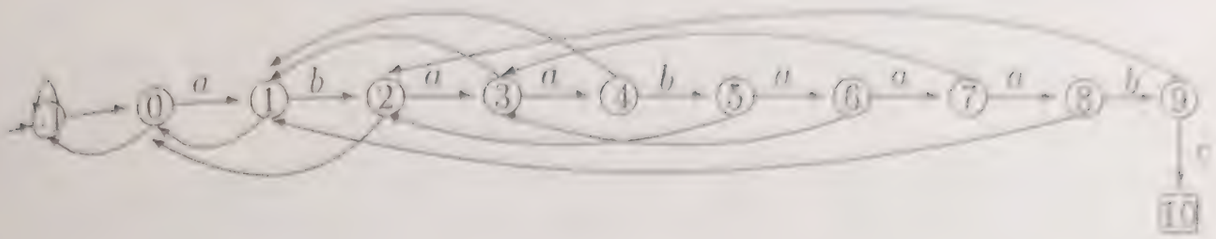
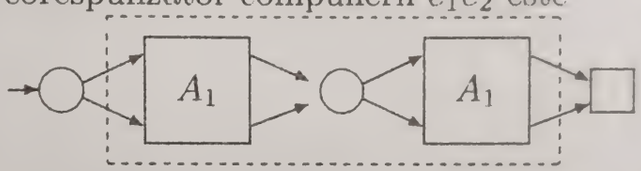



Figura 7.5: pattern şi funcţia *eşec* ca automat

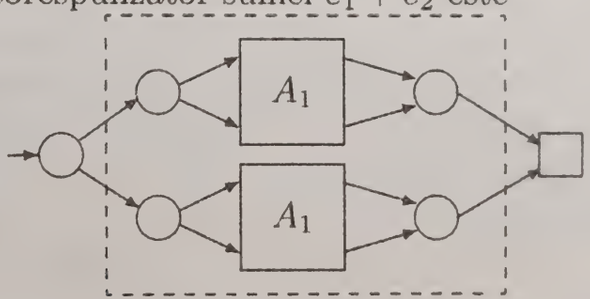
2. Fie $e_1$ şi $e_2$ două expresii regulate ale căror automate $A_1$ şi $A_2$ sunt reprezentate ca mai jos:

Atunci:

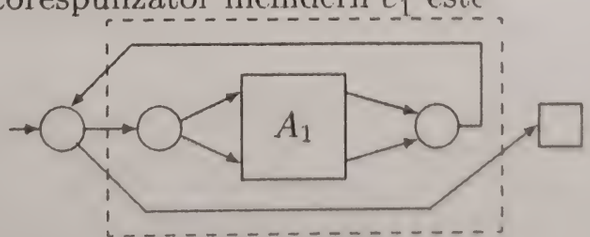
a) automatul corespunzător compunerii $e_1e_2$ este




b) automatul corespunzător sumei $e_1 + e_2$ este




c) automatul corespunzător închiderii $e_1^*$ este




**Exemplu.** Automatul corespunzător expresiei $e = a(b^*a + cd)$ este reprezentat în figura 7.6.a (câteva arce şi stări redondante au fost eliminate). sfex

Întâlnim două tipuri de stări (vârfuri în digraf) în definiţia unui automat asociat unei expresii: stări din care ies un singur arc etichetat cu un caracter şi stări din care ies două arce netichetate. Această proprietate este evidenţiată mai bine prin reprezentarea automatelor ca în figura 7.6.b. Notăm totuşi faptul că această proprietate rezultă din modul particular în care am construit automatul. În general, automatul asociat unei expresii nu este unic determinat, dar toate automatele corespunzătoare aceleaşi expresii sunt izomorfe (din punctul de vedere al

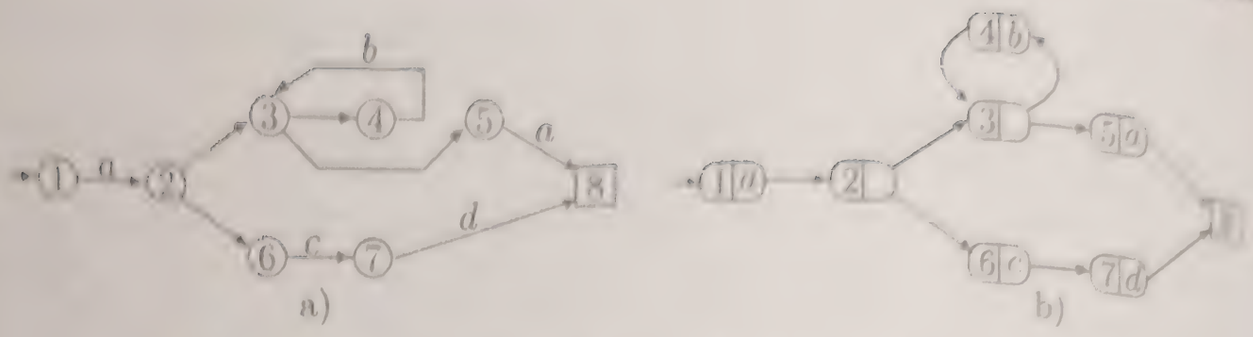



Figura 7.6: Automatul corespunzător expresiei $a(b^*a + cd)$

comportării). Reprezentarea particulară de aici permite reprezentarea automatelor printr-un tablou unidimensional de structuri, unde o componentă a tabloului are trei câmpuri: unul pentru memorarea caracterului, iar celelalte două pentru memorarea stărilor următoare. Facem convenţia ca $-1$ să reprezinte starea invalidă şi 0 starea iniţială. De exemplu, $A.stare[s].st\_urm2 = -1$ poate semnifica faptul că din starea $s$ pleacă un singur arc.

**Exerciţiul 7.5.1.** Să se scrie un program care construieşte automatul asociat unei expresii regulate.

Algoritmul de căutare a unui şir desemnat de o expresie regulată are următoarea descriere sumară. Utilizăm o coadă cu restricţii la ieşire $D$ (a se vedea şi exerciţiul 2.5.2), unde inserările se pot face şi la început şi la sfârşit, iar ştegerile/citirile numai la început. Fie $q$ starea curentă a automatului, $j$ poziţia curentă în textul $s$, $i$ poziţia în textul $s$ de început a patternului curent. Simbolul # va juca rolul de delimitator (el poate fi înlocuit cu starea invalidă $-1$). Iniţial avem $D = (\#), q = 1$ (prima stare după starea iniţială 0), $i = j = 1$. La pasul curent se execută următoarele operaţii:

1. Dacă:
   a) din $q$ pleacă două arce neetichetate, atunci inserează la început în $D$ destinaţiile celor două arce;
   b) din $q$ pleacă un singur arc etichetat cu $s[j]$, atunci inserează la sfârşitul lui $D$ destinaţia arcului;
   c) $q$ este delimitatorul #, atunci:
      i) dacă $D = \emptyset$, atunci incrementează $i$, $j$ devine noul $i$, inserează # în $D$ şi atribuie 1 lui $q$ (aceasta corespunde situaţiei când au fost epuizate toate posibilităţile de a găsi în text o apariţie a unui şir specificat de pattern care începe la poziţia $i$);
      ii) dacă $D \neq \emptyset$, atunci incrementează $j$ şi inserează # la sfârşitul lui $D$;
   d) dacă $q$ este starea finală, atunci s-a găsit o apariţie a unui şir specificat de pattern care începe la poziţia $i$.
2. Extrage starea de la începutul lui $D$ şi o memorează în $D$, după care reia pasul curent.

**Exerciţiul 7.5.2.** Să se exemplifice execuţia algoritmului de mai sus pentru automatul din figura 7.6 şi textul *abcacde*.

**Exerciţiul 7.5.3.** Să se scrie un program care implementează algoritmul de mai sus.

**Exerciţiul 7.5.4.** Fie $e$ o expresie regulată, $A$ automatul corespunzător lui $e$ şi $s$ un subiect.

1. Să se scrie un subprogram recursiv `PozSf(q,i):Integer` care, pentru o stare $q$ in $A$ şi o poziţie $i$ în subiect date, întoarce $-1$, dacă nu există drum de la $q$ la starea finală descris de un subşir al lui $s$ ce începe la poziţia $i$, şi poziţia de sfârşit al subşirului descris de un astfel de drum, atunci când există.
2. Să se scrie un program care utlizează `PozSf` pentru a determina o apariţie în $s$ a unui şir desemnat de $e$.
3. Să se modifice `PozSf(q,i)`, astfel încât să determine toate drumurile de la $q$ la starea finală descrise de subşiruri ale lui $s$ care încep la poziţia $i$.

## 7.6 Mai multe patternuri şi înlocuire

Considerăm două modificări ale problemei studiate în secţiunile anterioare: în loc de un singur pattern presupunem că se caută simultan pentru mai multe patternuri, iar apariţia fiecărui pattern este înlocuită cu un alt şir. Această nouă versiune este tratată în [AS84]. Notăm cu $K = \{(x_0, y_0), (x_2, y_2), \ldots, (x_{k-1}, y_{k-1})\}$ mulţimea perechilor (pattern, şir înlocuitor). Covenim ca şirurile $x_i$ şi $y_i$ să le numim cuvinte-cheie. Presupunem ($x_i \neq \varepsilon$), pentru orice $i$ cu $0 \leq i < k$. Căutarea simultană pentru mai multe chei şi înlocuirea acestora presupune rezolvarea problemei suprapunerii cheilor: când o cheie se suprapune peste altă cheie. În acest caz se va înlocui direct cheia mai lungă (evident, cele două chei au aceeaşi poziţie de start).

Algoritmul prezentat este o generalizare a algoritmului `KMP` şi presupune parcurgerea a trei etape. În prima etapă se memorează cheile într-o structură arborescentă asemănătoare arborilor digitali. O astfel de structură este reprezentată în figura 7.7 pentru

$$K = \{(ABCDE, \alpha), (CDE, \beta), (BC, \gamma)\}.$$

Vârfurile arborelui le numim stări (de la terminologia automatelor) şi le identificăm prin numere întregi nenegative. Mai considerăm în plus starea $-1$ care joacă rolul stării nedefinite. Rădăcina este starea 0 totdeauna. Arborele îl reprezentăm printr-o funcţie $g[s, c]$ cu semnificaţia $g[s, c] = s'$ dacă şi numai dacă există un arc de la starea $s$ la starea $s'$ etichetat cu caracterul $c$ (în terminologia automatelor, acţiunea notată de $c$ transformă starea $s$ în starea $s'$). Altfel $g[s, c] = Nedefinit$. Această reprezentare este foarte simplă, dar are dezavantajul că multe intrări într-un tabel au valoarea $Nedefinit$. Structura arborescentă o completăm cu o funcţie parţial definită $out[s]$ care, pentru o stare corespunzătoare terminării unei chei, înregistrează şirul înlocuitor. Pentru exemplul din figura 7.7 avem $out[5] = \alpha, out[8] = \beta, out[10] = \gamma$. Definiţia funcţiei $out$ urmează a fi completată în etapa a doua. Subprogramul care construieşte arborele utilizează o procedură de inserare a unei perechi de chei, similară celei de la arbori digitali:

```
procedure adaugaCheie(x, y)
    m ← lung(x)
    stare ← 0
    j ← 1
    while (g[stare,x[j]] ≠ nedefinit) do
        stare ← g[stare,x[j]]
        j ← j+1
    for i ← j to m do
        starenoua ← starenoua+1
        for c ← primulCaracter to ultimulCaracter do
            g[starenoua,c] ← nedefinit
            out[starenoua] ← NULL
        g[stare,x[i]] ← starenoua
        stare ← starenoua
    out[stare] ← y
end

procedure constrGraf(chei, k, g, out)
begin
    starenoua ← 0
    for c ← primulCaracter to ultimulCaracter do
        g[0,c] ← nedefinit
    for i ← 1 to k do
        adaugaCheie(chei[i,1],chei[i,2])
    for c ← primulCaracter to ultimulCaracter do
        if (g[0,c] = nedefinit) then g[0,c] ← 0
end
```

S-a presupus $chei[i,1] = x_i$ şi $Chei[i,2] = y_i$.

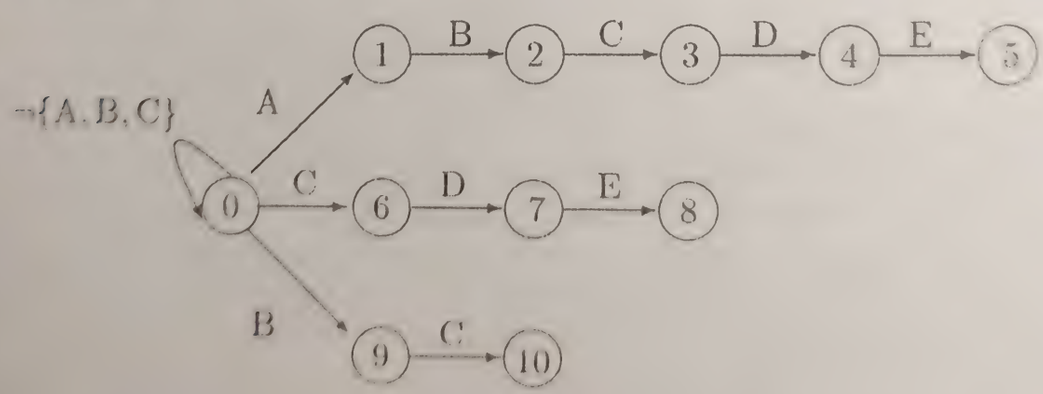



Figura 7.7: Arborele cheilor

În etapa a doua se construieşte funcţia eşec $f[s]$ cu semnificaţia că dacă apare nepotrivire în starea $s$, atunci următoarea stare va fi $f[s]$. Algoritmul care construieşte funcţia eşec este similar celui de la KMP, numai că acum ordinea de determinare a valorilor funcţiei este cea dată de explorarea BFS a arborelui. Funcţia eşec pentru exemplul de mai sus este reprezentată în figura 7.8. Tot în această etapă este

completată şi definiţia funcţiei *out* care va înregistra, pentru fiecare stare, şirul ce urmează a fi scris în textul rezultat, corespunzător acelei stări. Pentru stările $s = 0, i$ punem $out[s] = 'c'$. Pentru exemplul nostru avem $out[1] = 'A'$, $out[6] = 'C'$ şi $out[9] = 'B'$. În continuare, funcţia *out* se calculează odată cu funcţia eşec. Avem $out[2] = out[3] = 'A'$, $out[7] = 'CD'$, $out[4] = 'A\gamma D'$. Valorile $out[5] = \alpha$, $out[8] = \beta$ şi $out[10] = \gamma$ au fost calculate la prima etapă. Programul care realizează etapa a doua este următorul:

```
procedure constrFctEsec(g, out, f)
   q ← coadaVida()
   for c ← primulCaracter to ultimulCaracter do
       stare1 ← g[0,c]
       if ((stare1 ≠ 0) and (stare1 ≠ nedefinit))
          then insereaza(q,stare1)
               f[stare1] ← 0
               if (out[stare1] = NULL)
                  then new(out[stare1])
                       out[stare1] ← c
   while (not esteCoadaVida(q)) do
      stare3 ← citeste(q)
      elimina(q)
      for c ← primulCaracter to ultimulCaracter do
          stare1 ← g[stare3,c]
          if ((stare1 ≠ Nedefinit) and (stare1≠0))
             then insereaza(q, stare1)
                  if out[stare1] ≠ NULL
                     then f[stare1] ← 0
                     else stare2 ← f[stare3]
                  out[stare1] ← out[stare3]
                  while (g[stare2,c] = nedefinit) do
                     stare2 ← f[stare2]
                     concatenate(out[stare1], out[stare2])
                  f[stare1] ← g[stare2,c]
                  if (f[stare1] = 0)
                     then concatenate(out[stare1],c)
end
```

A treia etapă constă în procesarea textului, găsirea şi înlocuirea cheilor.

```
procedure MPMR(g, out, f, s, n)
   stare ← 0
   for i ← 1 to n do
       while (g[stare,s[i]] = nedefinit) do
          scrie(out[stare])
          stare ← f[stare]
       stare ← g[stare,s[i]]
       if (stare = 0) then scrie(s[i])
       while (stare ≠ 0) do
```

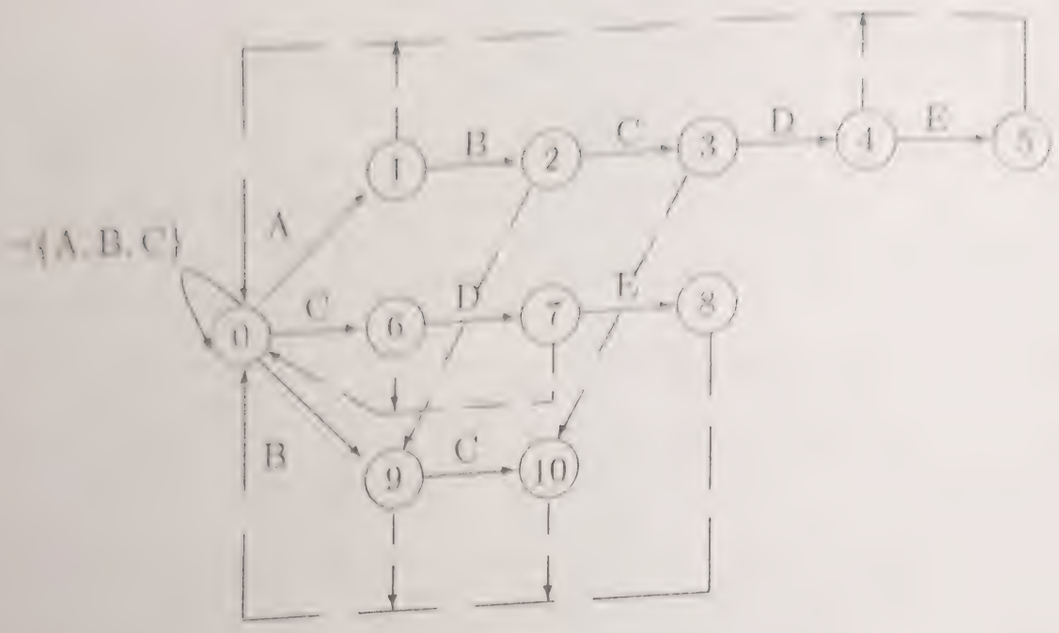



Figura 7.8: Funcţia eşec

```
        scrie(out[stare])
        stare ← f[stare]
    end
```

**Exerciţiul 7.6.1.** Să se aplice algoritmul MPMR pentru textul 'BABCDEXABCAB' și mulțimea de chei din exemplu.

Algoritmii de preprocesare necesită un timp proporţional cu suma lungimilor cheilor, iar algoritmul de căutare şi înlocuire realizează cel mult $2n$ tranziţii de stări [AS84].

## 7.7 Exerciţii

**Exerciţiul 7.1.** Să se implementeze un algoritm de căutare naivă care să cerceteze patternul de la dreapta la stânga.

**Exerciţiul 7.2.** Să se modifice algoritmii de pattern matching pentru cazul în care patternul conţine un caracter *wildcard* (orice caracter din text se potriveşte peste el).

**Exerciţiul 7.3.** Să se implementeze o versiune a algoritmului Rabin-Karp pentru *pattern*uri în texte bidimensionale. Se presupune că atât patternul, cât şi textul sunt dreptunghiuri de caractere.

**Exerciţiul 7.4.** Să se projecteze un algoritm liniar care determină dacă un text $T$ este o rotaţie ciclică a altui text $T'$. De exemplu, tara şi rata sunt rotaţii ciclice unul altuia.

**Exerciţiul 7.5.** ([CLR93]) Cu $w^i$ notăm concatenarea şirului $w$ cu el însuşi de $i$ ori. De exemplu, $(abc)^4 = abcabcabcabc$. Numim *factor de repetiţie* al şirului $w$ un număr $r$ cu $w = u^r$, unde $u$ este un şir şi $r > 0$. Notăm cu $\rho(w)$ cel mai mare factor de repetiţie al şirului $w$. Să se proiecteze un algoritm eficient care, pentru un şir $w[1..n]$ dat, determină $\rho(w[1..i])$ pentru $i = 1, 2, \ldots, n$.

**Exerciţiul 7.6.** Să se modifice algoritmul de pattern-matching-simplu, astfel încât să fie determinate toate aparitiile patternului în text şi să fie înlocuite cu un alt şir dat.

## 7.8 Referinţe bibliografice

Acest capitol, exceptând secţiunea dedicată patternurilor multiple, urmează linia din [Sed88]. Patternurile multiple şi înlocuirea sunt tratate în [AS84]. Mai pot fi consultate [Baa78, CLR93, AHU74].

# Capitolul 8

# Despre paradigmele de proiectare

## 8.1 Aspecte generale

Descrierea unui algoritm care rezolvă o problemă presupune ca etapă intermediară construcţia modelului matematic corespunzător problemei (a se vedea figura 8.1). Necesitatea construcţiei modelului matematic este impusă de următoarele motive:

- *eliminarea ambiguităţilor şi inconsistenţelor.* De multe ori, problema este descrisă informal (verbal). De aici, anumite aspecte ale problemei pot fi omise sau formulate ambiguu. Construcţia modelului matematic evidenţiază toate aceste lipsuri şi, în acest fel, conduce la eliminarea lor;
- *utilizarea instrumentelor matematice de investigare.* Parcurgerea drumului de la formularea problemei la găsirea soluţiei nu este întotdeauna simplă sau predictabilă. Instrumentele matematice de investigare oferă un cadru sistematic şi sigur pentru determinarea structurii analitice a soluţiei şi apoi a modului de obţinere a acesteia;
- *diminuarea efortului la scrierea programului.* Descrierea soluţiei în termenii modelului matematic uşurează foarte mult munca de definire şi apoi de descriere a algoritmului (programul).

O paradigmă de proiectare a algoritmilor se bazează pe un anumit tip de model matematic şi pune la dispoziţie procedee prin care se poate construi şi implementa (descrie ca program) un model particular corespunzător unei probleme. Descrierea unui model matematic indexmodel matematic cuprinde următoarele trei aspecte [BD62]:

1. *conceptual*: presupune identificarea şi definirea conceptelor care descriu componentele din domeniul problemei;
2. *analitic*: presupune găsirea tuturor relaţiilor între concepte care conduc la găsirea şi descrierea soluţiei;
3. *computaţional*: presupune evaluarea calitativă a algoritmului ce construieşte soluţia.

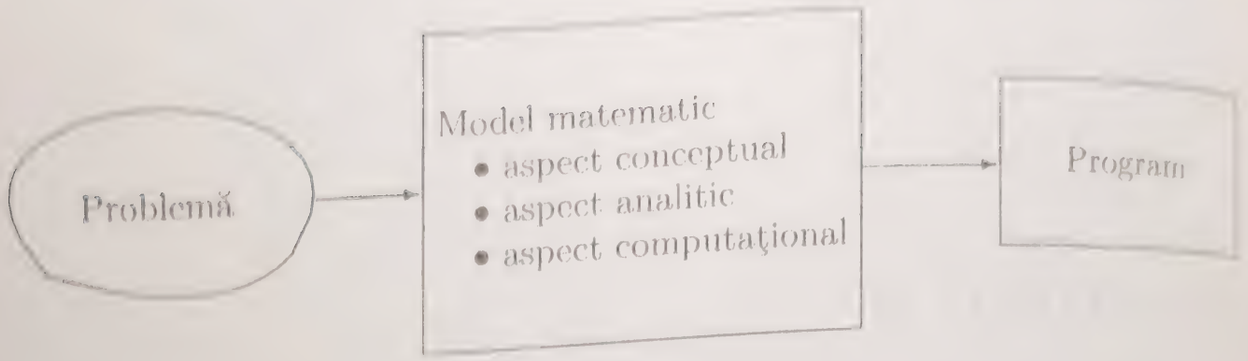



Figura 8.1: Drumul de la problemă la program

Cele trei aspecte se reflectă în etapele ce trebuie parcurse în rezolvarea unei probleme şi pe care le-am discutat în capitolul de introducere.

Următoarele capitole prezintă cele mai cunoscute paradigme de proiectare: *greedy*, *divide-et-impera*, programare dinamică, *backtracking* şi *branch-and-bound*. Pentru fiecare paradigmă sunt prezentate modelul matematic şi un set de studii de caz. În acest capitol prezentăm, ca exemplu, paradigma eliminării pentru a evidenţia aspectele enumerate mai sus.

## 8.2 Un exemplu simplu de paradigmă: eliminarea

### 8.2.1 Primul studiu de caz: problema celebrităţii

#### 8.2.1.1 Descrierea problemei

Se consideră o colectivitate formată din $n$ persoane. O celebritate este o persoană care este cunoscută de oricare altă persoană şi nu cunoaşte pe nimeni. Problema constă în a scrie un program care să determine dacă există o celebritate în cadrul colectivităţii prin întrebări de forma: „Persoana $i$ cunoaşte persoana $j$?". Programul va pune un număr cât mai mic de întrebări.

#### 8.2.1.2 Modelul matematic

**Aspectul conceptual.** Fără să restrângem generalitatea, presupunem că persoanele colectivităţii sunt identificate prin numerele $0, 1, 2, \ldots, n-1$. O întrebare de forma „Persoana $i$ cunoaşte persoana $j$?" va fi reprezentată de următoarea funcţie:

$$\texttt{cunoaste} : \{0, 1, 2, \ldots, n-1\} \times \{0, 1, 2, \ldots, n-1\} \to \{\texttt{true}, \texttt{false}\}$$

**Aspectul analitic.** Notăm cu CAND mulţimea persoanelor care pot candida la calitatea de celebritate. Iniţial, orice persoană poate candida la această calitate. După primirea răspunsului la o întrebare există următoarele două posibilităţi:

- Răspunsul este adevărat, *i.e.*, persoana $i$ cunoaşte pe $j$. Atunci $i$ nu poate fi celebritate şi poate fi eliminat din CAND.
- Răspunsul este fals. Atunci $j$ nu poate fi celebritate şi deci poate fi eliminat din CAND.

În ambele cazuri se observă că este eliminat un posibil candidat. Procesul de eliminare poate fi repetat până când rămâne un singur candidat.

**Aspectul computaţional.** Obţinerea unui singur candidat se poate realiza prin $n-1$ întrebări. Notăm că nu putem trage concluzia încă dacă singurul candidat rămas este celebritate sau nu. Mai trebuie puse întrebări suplimentare pentru a afla acest lucru. Această verificare se poate face prin cel mult $2\cdot(n-1)$ întrebări. Rezultă că determinarea celebrităţii se poate face punând cel mult $3 \cdot (n-1)$ întrebări.

#### 8.2.1.3 Implementare

Deoarece la sfârşit CAND va avea un singur element, ca poate fi gestionată de o variabilă de tip întreg `cand`. Se pleacă de la ipoteza că persoana candidat este 0 şi apoi se pun întrebări între posibilul candidat şi persoana următoare pentru a afla noul candidat. Pentru ultimul candidat se verifică dacă este celebritate prin întrebări duble între el şi fiecare dintre celelalte $n-1$ persoane.

Funcţia `Celebritate` întoarce valoarea $-1$, dacă nu există celebritate; altfel întoarce numărul corespunzător celebrităţii.

```
function celebritate(n)
   cand ← 0
   for j ← 1 to n-1 do
       if (cunoaste(i, j))
          then cand ← j
   i ← 0
   while (cand >= 0) and (i < n) do
      if (not cunoaste(i, cand) or cunoaste(cand, i))
         then cand ← -1
      i ← i+1
   return cand
end
```

#### 8.2.1.4 Exerciţii

**Exerciţiul 8.2.1.** În timpul execuţiei subprogramului `Celebritate`, în partea care verifică dacă unicul candidat este celebritate este posibil ca anumite întrebări să se repete, ele mai fiind puse la determinarea candidatului. Este posibilă atât eliminarea acestei redundanţe cât şi păstrarea timpului de execuţie şi a spaţiului de memorie utilizat?

**Exerciţiul 8.2.2.** Să se găsească similarităţi între problema celebrităţii şi cea din exerciţiul 2.9.10.

### 8.2.2 Al doilea studiu de caz: permutarea maximală

#### 8.2.2.1 Prezentarea problemei

Se consideră o colectivitate formată din $n$ persoane. Fiecare persoană din colectivitate are un singur preferat în colectivitate. Este posibil ca o persoană să se poate

prefera pe sine însăşi. Un grup de persoane din colectivitate se numeşte *omogen* dacă fiecare membru al grupului este preferatul exact al unui singur membru din acel grup. Se pune problema alegerii unui grup omogen cu număr maxim de persoane.

### 8.2.2.2 Modelul matematic

**Aspectul conceptual.** Notăm cu $A$ mulţimea finită reprezentând colectivitatea de persoane şi cu $f : A \to A$ funcţia care reprezintă relaţia de preferinţă: $f(i) = j$ înseamnă că $j$ are ca preferat pe $i$ (sau, echivalent, $i$ este preferat de $j$). Problema constă în a determina o submulţime $X$ a lui $A$ cu proprietăţile:

i) $f(X) \subseteq X$ (cei care preferă persoane din $X$ se află de asemenea în $X$);
ii) $f|_X$ este injectivă (orice membru al grupului are un singur preferat);
iii) $X$ are număr maxim de elemente peste clasa submulţimilor lui $A$ care satisfac i) şi ii).

Prin $f|_X$ am notat restricţia funcţiei $f$ la submulţimea $X$. Remarcăm faptul că primele două condiţii, i) şi ii), sunt echivalente cu faptul că $f|_X : X \to X$ este bijecţie (permutare).

**Aspectul analitic.** Vom aplica o strategie asemănătoare cu cea din problema celebrităţii: se vor elimina pe rând din $A$ elementele nedorite până când se va obţine $X$. Rămâne de determinat criteriul de eliminare. Pornim cu analiza problemei iniţiale. Distingem următoarele două cazuri:

- $f(A) = A$. Atunci $X = A$.
- există $j \in A$ cu proprietatea că pentru orice $i \in A$, $f(i) \neq j$. Problema $P'$ cu instanţa $A' = A \setminus \{j\}$, $f|_{A'}$ şi problema iniţială au aceeaşi soluţie.

Aşadar, putem porni cu $\mathrm{CAND} = A$ şi la fiecare pas să eliminăm un $j$ care nu aparţine imaginei funcţiei $f|_{\mathrm{CAND}}$. Procesul de eliminare continuă atât cât este posibil şi la sfârşit vom avea $X = \mathrm{CAND}$.

**Aspectul computaţional.** Vom arăta mai întâi că testul de eliminare pentru un $j$ fixat poate fi efectuat în timpul $O(1)$. Presupunem, fără să restrângem generalitatea, că $A = \{0, 1, \ldots, n-1\}$. Mulţimea CAND poate fi reprezentată prin vectorul caracteristic:

$$x[i] = \begin{cases} 1, & \text{dacă } i \in \mathrm{CAND} \\ 0, & \text{altfel} \end{cases}$$

Cu o mică modificare, vectorul x poate fi utilizat şi pentru reprezentarea funcţiei $f|_{\mathrm{CAND}}$:

$$x[i] = \begin{cases} f(i), & \text{dacă } i \in \mathrm{CAND} \\ -1, & \text{altfel} \end{cases}$$

Cu această reprezentare, verificarea condiţiei $(\exists i \in \mathrm{CAND})\ f(i) = j$, pentru un $j$ fixat, necesită în cazul cel mai nefavorabil $n$ comparaţii. Acest număr poate fi redus la 1 dacă vom considera un tablou suplimentar $b$ cu următoarea semnificaţie:

$$b[j] = \#\{i \in \mathrm{CAND} \mid f(i) = j\}$$

Acum, verificarea condiţiei $f(\mathrm{CAND}) = \mathrm{CAND}$ se poate realiza printr-o singură parcurgere a tabloului $b$: dacă există $j$ cu $x[j] \geq 0$ şi $b[j] = 0$, atunci $j$ va fi eliminat din CAND. Rezultă că problema determinării permutării maximale poate fi rezolvată în timpul $O(n^2)$.

#### 8.2.2.3 Implementare

Programul `permMax` are următoarea descriere:

```
procedure permMax(f, n, x)
   for j ← 0 to n-1 do
       b[j] ← 0
   for i ← 0 to n-1 do
       b[f[i]] ← b[f[i]] + 1
   for i ← 0 to n-1 do
       x[i] ← f[i]
   existaj ← true
   while (existaj) do
      existaj ← false
      j ← 0
      while ((j < n) and not existaj) do
      if ((x[j] ≥ 0) and (b[j] = 0))
         then existaj ← true
              x[j] ← -1
              b[f[j]] ← b[f[j]] - 1
      j ← j+1
end
```

#### 8.2.2.4 Exerciţii

**Exerciţiul 8.2.3.** Funcţia $f$ poate fi reprezentată printr-un digraf: există arc de la $i$ la $j$ dacă şi numai dacă $f(i) = j$. Să se arate că determinarea permutării maximale este echivalentă cu determinarea tuturor circuitelor în digraful asociat.

**Exerciţiul 8.2.4.** Să se proiecteze un algoritm care să determine toate circuitele în digraful asociat funcţiei $f$. Să se precizeze complexităţile timp şi spaţiu pentru algoritmul proiectat.

**Exerciţiul 8.2.5.** Poate fi utilizat algoritmul de la exerciţiul 8.2.4 la determinarea tuturor circuitelor într-un digraf oarecare?

### 8.2.3 Prezentarea generală a paradigmei

Strategiile aplicate în studiile de caz precedente sunt foarte asemănătoare. Aici vom arăta cum aceste strategii pot fi generalizate la nivel de paradigmă.

#### 8.2.3.1 Modelul matematic

**Aspectul conceptual.** Problemele pentru care poate fi aplicată strategia eliminării sunt modelate după următoarea schemă:

> Se consideră o mulţime $S$ cu $n$ elemente. Se pune problema determinării existenţei unei submulţimi maximale $A \subseteq S$ care satisface o proprietate $P(A)$ (când există).

**Aspectul analitic.** Se caută o condiţie $C$ cu proprietatea $C(x) \Rightarrow x \notin A$. Fie CAND submulţimea obţinută din $S$ prin eliminarea elementelor $x$ ce nu satisfac $C(x)$. Apoi se verifică dacă submulţimea CAND satisface proprietatea $P$ şi dacă da, atunci $A = \text{CAND}$.

**Aspectul computaţional.** Presupunem că verificarea condiţiei $C(x)$ se face în timpul $O(n^p)$. Determinarea submulţimii de candidaţi CAND se face în timpul $O(n^{p+1})$. Dacă verificarea proprietăţii $P$ se face în timpul $O(n^q)$, atunci determinarea lui $A$ necesită timpul $O(n^{p+1} + n^q)$.

#### 8.2.3.2 Implementare

Programul care descrie soluţia dată de strategia eliminării este foarte simplu şi are următoarea descriere schematică:

```
function sol(S)
   CAND ← S
   for each x din CAND do
       if C(x) // x satisface criteriul de eliminare
          then CAND ← CAND \ {x}
   if P(CAND) // CAND satisface P
      then return CAND
      else return ∅
end
```

## 8.3 Alte consideraţii privind paradigmele de proiectare

Aşa cum s-a observat deja din studiul de caz precedent, construcţia modelului matematic nu se poate face totdeauna cu o delimitare netă între cele trei aspecte conceptual, analitic şi computaţional. De fapt, în [BD62] nici nu se recomandă un astfel de delimitare. Este mult mai natural şi mai eficient ca toate cele trei aspecte să fie dezvoltate simultan. Este posibil ca dezvoltarea relaţiilor analitice între concepte să evidenţieze anumite aspecte ale problemei care nu au fost complet sau corespunzător conceptualizate. În acest caz o revizuire a conceptelor este necesară. De asemenea, anumite performanţe computaţionale pot fi îmbunătăţite prin găsirea unor reprezentări echivalente conceptual, dar mai performante.

De multe ori nici cele două faze ale rezolvării problemei – modelul matematic şi implementarea – nu pot fi considerate separat. O implementare defectuoasă poate reduce performanţele soluţiei descrise de model sau, dimpotrivă, o implementare bună poate îmbunătăţi performanţele soluţiei analitice. În alte cazuri, aspectul computaţional poate fi realizat numai după descrierea implementării.

## 8.4 Referinţe bibliografice

Consideraţiile despre necesitatea construirii modelului matematic, ca pas intermediar pe drumul problemă – program, se bazează pe cele din [BD62].

Formalizarea tehnicii de eliminare ca metodă apare pentru prima dată aici. Cele două probleme apar ca exemple independente în [Man89]. Problema celebrităţii, formalizată în limbajul teoriei grafurilor, apare ca exerciţiu în [CLR93, CLR00, MS91, Baa78, BG00]. Testarea primalităţii unui număr întreg utilizând testul Fermat [Luc93] constituie un alt exemplu de algoritm bazat pe tehnica eliminării.

# Capitolul 9

# Algoritmi *greedy*

## 9.1 Memorarea eficientă a programelor

### 9.1.1 Descrierea problemei

$N$ programe (fişiere) urmează a fi memorate pe o bandă magnetică. Citirea unui program presupune citirea tuturor programelor aflate înaintea sa şi deci timpul de regăsire este suma timpilor de citire ai tuturor acestor programe (inclusiv cel căutat). *Timpul mediu* de regăsire este media aritmetică a celor $n$ timpi de regăsire. Problema constă în găsirea unei aranjări a celor $n$ programe astfel încât timpul mediu de regăsire să fie minim.

### 9.1.2 Modelul matematic

Problema poate fi descrisă în următoarea manieră mai abstractă. Considerăm $n$ obiecte notate cu $0, 1, \ldots, n-1$ de dimensiuni $L_0, \ldots, L_{n-1}$, respectiv, care urmează a fi aranjate într-o listă liniară. Dacă obiectele sunt aranjate în ordinea $(\pi(0), \ldots, \pi(n-1))$, atunci timpul $t_k$ de regăsire al obiectului $\pi(k)$ (care se află pe poziţia $k$) este $\sum_{j=0}^{k} L_{\pi(j)}$ şi timpul mediu de regăsire al tuturor obiectelor este $TM = \frac{1}{n}\sum_{k=0}^{n-1} t_k$. Problema constă în determinarea unei permutări $\pi = (\pi(0), \ldots, \pi(n-1))$ astfel încât, aranjând obiectele în ordinea dată de $\pi$, timpul mediu de regăsire să fie minim.

Scriem $SUMA(\pi) = \sum_{k=0}^{n-1} t_k$ detaliat:

$$\begin{aligned} SUMA(\pi) = & L_{\pi(0)} + && (k=0)\\ & L_{\pi(0)} + L_{\pi(1)} + && (k=1)\\ & L_{\pi(0)} + L_{\pi(1)} + L_{\pi(2)} + && (k=2) \end{aligned}$$

$$\dots$$
$$L_{\pi(0)} + L_{\pi(1)} + \cdots + L_{\pi(n-1)} \qquad (k = n-1)$$

Timpul mediu depinde direct de această sumă. Este uşor de văzut că obiectele de la începutul listei contribuie de mai multe ori la sumă. Intuiţia ne spune că suma este cu atât mai mică cu cât elementele de la începutul listei sunt mai mici. De aici, rezultă următoarea strategie de aranjare a obiectelor în lista liniară:

- dacă până la momentul $i-1$ s-a construit deja permutarea parţială $(\pi(0), \dots, \pi(i-1))$, atunci la momentul $i$ se va alege obiectul $\pi(i) = k$ cu dimensiunea $L_k$ minimă peste mulţimea obiectelor nearanjate în listă.

Corectitudinea strategiei este dată de următoarea teoremă:

**Teorema 9.1.** *Dacă* $L_{\pi(0)} \le L_{\pi(1)} \le \cdots \le L_{\pi(n-1)}$, *atunci*

$$SUMA(\pi) = \min\{SUMA(\pi') \mid \pi' \text{ o permutare a mulţimii } \{0, 1, \dots, n-1\}\}$$

*Demonstraţie.* Aplicăm metoda reducerii la absurd. Fie $\pi$ o permutare optimă pentru care există $i < j$ cu $L_{\pi(i)} > L_{\pi(j)}$. Notăm cu $\pi'$ permutarea $\pi$ înmulţită cu transpoziţia $(i, j)$. Făcând calculele, se obţine $SUMA(\pi) > (SUMA(\pi')$. Contradicţie. sfdem

### 9.1.3 Implementare

Algoritmul corespunzător strategiei de mai sus este descris de schema procedurală `memprog`:

```
procedure memprog(L, n, π)
begin
   S ← {0,...,n − 1}
   while (S ≠ ∅) do
      alege k ∈ S astfel încît L[k] = min{L[j] | j ∈ S}
      S ← S \ {k}
      π[i] ← k
end
```

O preprocesare care ordonează a obiectele în ordinea crescătoare a dimensiunilor conduce la alegerea obiectului $k$ în timpul $O(1)$. Rezultă că problema poate fi rezolvată în timpul cel mai nefavorabil $O(n \log n)$.

## 9.2 Prezentare intuitivă a paradigmei

Principalele ingrediente ale paradigmei algoritmilor *greedy* sunt următoarele:

1. $S$ – o mulţime de intrări;
2. $\mathcal{C}$ – un tip de date cu proprietăţile:
   a) obiectele din $\mathcal{C}$ reprezintă submulţimi ale lui $S$;
   b) operaţiile includ inserarea $(X \cup \{x\})$ şi eliminarea $(X \setminus \{x\})$.

3. *Clasa de probleme* la care se aplică include probleme de optim. Convenim să luăm ca exemplu următoarea descriere schematică:
   *Intrare*: $S$;
   *Ieşire*: O submulţime maximală $B$ din $\mathcal{C}$ care optimizează o funcţie $f$ cu valori reale.
   În subsecţiunea 9.10 vom vedea că trebuie adăugate anumite condiţii suplimentare. Pentru moment, această descriere sumară este suficientă pentru scopul nostru.
   Pentru problema memorării programelor, considerăm $S = \{(i,j) \mid 0 \leq i,j < n\}$, $X \in \mathcal{C}$ dacă şi numai dacă $(\forall (i,j),(i',j') \in X)\, i = i' \iff j = j'$. O pereche $(i,j)$ are semnificaţia „pe poziţia $i$ se află obiectul $j$". Cu alte cuvinte, presupunem că la intrare avem toate posibilităţile în care putem aranja obiectele. O submulţime $X \in \mathcal{C}$ corespunde unei aranjări parţiale: pe o poziţie $i$ se află un singur obiect (relaţia $X$ este funcţională) şi un obiect $j$ se află pe o singură poziţie (relaţia $X$ este injectivă). O submulţime maximală $B$ desemnează o permutare $\pi$. Funcţia $f$ peste submulţimi maximale este *SUMA*. Ea poate fi extinsă la mulţimi arbitrare $X$ prin $f(X) = \sum(t_k(X) \mid (\exists \ell)\,(k,\ell) \in X)$, unde $t_k(X) = \sum(L_j \mid (\exists i)\,(i,j) \in X \wedge j \leq k)$.
4. Paradigma se bazează pe următoarele două proprietăţi:
   a) *Proprietatea de alegere locală.* Soluţia problemei se obţine făcând alegeri optime locale (de aici şi denumirea de *greedy*=„lacom"). O alegere optimă locală poate depinde de alegerile de până atunci, dar nu şi de cele viitoare. Alegerile optime locale nu asigură automat că soluţia finală realizează optimul global, adică constituie o soluţie a problemei. Trebuie demonstrat acest fapt. De regulă, aceste demonstraţii nu sunt foarte simple. Acesta este un inconvenient major al metodei *greedy*. Algoritmii sunt relativ simpli, dar demonstrarea faptului că aceştia rezolvă într-adevăr problema de optim asociată este deseori dificilă. În subsecţiunea 9.10 vom prezenta un cadru formal pentru aceste demonstraţii.
   O alegere optimă locală (alegere *greedy*) pentru problema memorării programelor constă în selectarea la pasul $i$ a unei perechi $i \mapsto j$ cu proprietate că $L_j$ este minim peste $\{L_{j'} \mid j' \text{ neales}\}$. Observăm că această alegere depinde numai de alegerile făcute până atunci. În final a trebuit să demonstrăm că aceste alegeri produc o permutare optimă.
   b) *Proprietatea de substructură optimă.* Soluţia optimă a problemi conţine soluţiile optime ale subproblemelor.
   Pentru problema memorării programelor, dacă $B$ este o permutare optimă, atunci orice $B' \subseteq B$ este o permutare optimă pentru submulţimea de obiecte $i$ cu $i \mapsto j \in B'$.

## 9.3 Arbori ponderaţi pe frontieră optimi

Studiul de caz prezentat în această secţiune are un grad de generalitate destul de mare astfel că poate fi considerat mai degrabă ca fiind o subparadigmă. Vom vedea că multe dintre problemele prezentate în secţiunile următoare sunt de fapt cazuri

particulare de arbori ponderați pe frontieră optimi. Din acest motiv, prezentarea este făcută direct la nivelul abstract.

### 9.3.1 Modelul matematic

Considerăm arbori binari cu proprietatea că orice vârf are 0 sau 2 succesori, iar vârfurile de pe frontieră au ca informații (etichete, ponderi) numere reale. Fie $inf(v)$. Convenim să numim acești arbori ca fiind *ponderați pe frontieră*. Pentru un vârf $v$ din arborele $t$ notăm cu $d_v$ lungimea drumului de la rădăcina lui $t$ la vârful $v$. *Lungimea externă ponderată* a arborelui $t$ este:

$$\text{LEP}(t) = \sum_{v \text{ pe frontiera lui } t} d_v \cdot inf(v)$$

Modificăm acești arbori etichetând vârfurile interne cu numere ce reprezintă suma etichetelor din cele două vârfuri fii. Adică, pentru orice vârf intern $v$ avem $inf(v) = inf(v_1) + inf(v_2)$, unde $v_1, v_2$ sunt fiii lui $v$ (a se vedea și figura 9.1).

**Lema 9.1.** *Fie $t$ un arbore binar ponderat pe frontieră. Atunci*

$$\text{LEP}(t) = \sum_{v \text{ intern în } t} inf(v)$$

*Demonstrație.* Se procedează prin inducție după $n$, numărul de vârfuri de pe frontiera lui $t$.

*Baza inducției.* Presupunem $n = 2$. Relația este evidentă.

*Pasul inductiv.* Presupunem că $t$ are $n+1$ vârfuri pe frontieră. Fie $v_1$ și $v_2$ două vârfuri de pe frontieră cu același predecesor imediat (tată) $v_3$. Avem $d_{v_1} = d_{v_2} = d$ și $inf(v_3) = inf(v_1) + inf(v_2)$. Considerăm arborele $t'$ obținut din $t$ prin eliminarea vârfurilor $v_1$ și $v_2$. Acum vârful $v_3$ se află pe frontiera lui $t'$. Conform ipotezei inductive avem:

$$\text{LEP}(t') = \sum_{v \text{ intern în } t'} inf(v) \tag{9.1}$$

Utilizăm (9.1) pentru a calcula lungimea externă ponderată a lui $t$:

$$\begin{aligned}
\text{LEP}(t) &= \sum_{v \text{ pe frontiera lui } t} d_v \cdot inf(v) \\
&= \sum_{v \text{ pe frontiera lui } t, v \neq v_1, v_2} d_v \cdot inf(v) + d \cdot inf(v_1) + d \cdot inf(v_2) \\
&= \sum_{v \text{ pe frontiera lui } t, v \neq v_1, v_2} d_v \cdot inf(v) + (d-1)(inf(v_1) + inf(v_2)) + inf(v_3) \\
&= \sum_{v \text{ pe frontiera lui } t, v \neq v_1, v_2} d_v \cdot inf(v) + (d-1) inf(v_3) + inf(v_3) \\
&= \sum_{v \text{ pe frontiera lui } t'} d_v \cdot inf(v)
\end{aligned}$$

$$= \mathrm{LEP}(t') + inf(v_3)$$
$$= \sum_{v \text{ intern în } t'} inf(v) + inf(v_3)$$
$$= \sum_{v \text{ intern în } t} inf(v)$$

S-a ţinut cont de faptul că interiorul lui $t$ este format din interiorul lui $t'$ la care se adaugă $v_3$.

**Exemplu.** Fie arborele din figura 9.1. Lungimea externă ponderată a arborelui

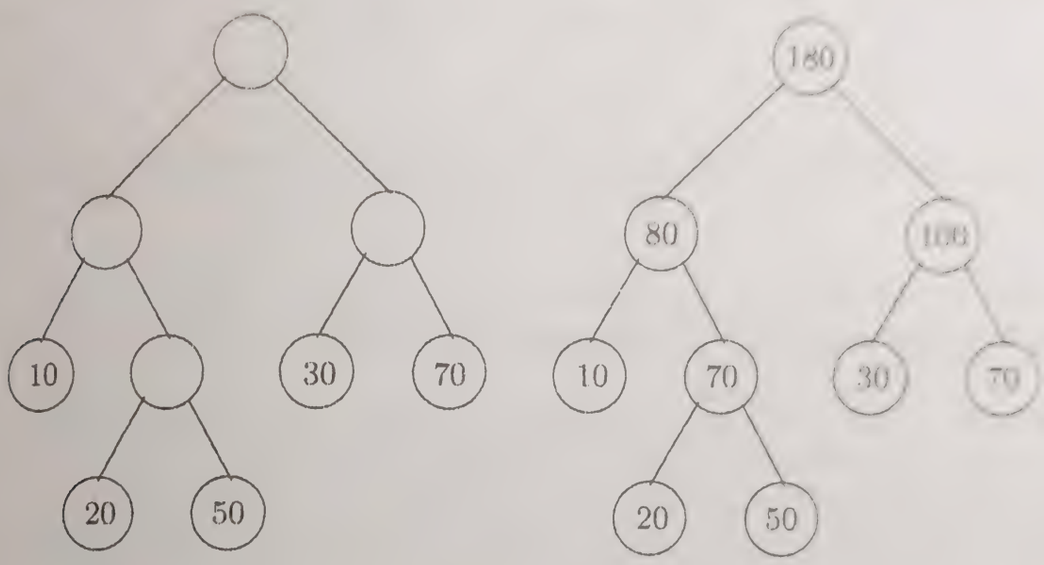



Figura 9.1: Arbore ponderat pe frontieră, înainte şi după modificare

este egală cu $80+70+180+100 = 10\cdot 2+20\cdot 3+50\cdot 3+30\cdot 2+70\cdot 2 = 430$.

În continuare ne ocupăm de următoarea problemă:

Fie dată $x = (x_0, \ldots, x_{n-1})$ o secvenţă (listă liniară) de numere. Problema constă în determinarea unui arbore binar ponderat pe frontieră care are ca informaţii în cele $n$ vârfuri de pe frontieră numerele $x_0, \ldots, x_{n-1}$ şi cu lungimea externă ponderată minimă.

O metodă total ineficientă ar putea fi generarea tuturor arborilor binari cu $n$ vârfuri pe frontieră etichetate cu elementele secvenţei $x$ şi alegerea unuia cu lungimea externă ponderată minimă.

**Exerciţiul 9.3.1.** Să se arate că mulţimea arborilor binari ponderaţi pe frontieră cu elementele secvenţei $x$ este finită.

Pentru a obţine un algoritm eficient este necesar să studiem mai mult proprietăţile arborilor cu LEP minimă. Prin $\mathcal{T}(x)$ notăm mulţimea arborilor binari care au ca informaţii în vârfurile de pe frontieră numerele din secvenţa $x$.

**Lema 9.2.** *Fie $t$ un arbore din $\mathcal{T}(x)$ cu LEP minimă şi $v_1, v_2$ două vârfuri pe frontiera lui $t$. Dacă $inf(v_1) < inf(v_2)$ atunci $d_{v_1} \geq d_{v_2}$.*

*Demonstraţie.* Presupunem $d_{v_1} < d_{v_2}$. Notăm $d_1 = d_{v_1}$ şi $d_2 = d_{v_2}$. Fie $t'$ arborele obţinut din $t$ prin interschimbarea vârfurilor $v_1$ şi $v_2$. Avem:

$$\begin{aligned}
\mathrm{LEP}(t') &= \sum_{v \text{ pe frontiera lui } t'} d_v \cdot inf(v) \\
&= \sum_{v \text{ pe frontiera lui } t', v \neq v_1, v_2} d_v \cdot inf(v) + d_1 \cdot inf(v_2) + d_2 \cdot inf(v_1) \\
&= \sum_{v \text{ pe frontiera lui } t} d_v \cdot inf(v) - d_1 \cdot inf(v_1) - d_2 \cdot inf(v_2) + d_1 \cdot inf(v_2) + \\
&\qquad d_2 \cdot inf(v_1) \\
&= \mathrm{LEP}(t) - (d_1 - d_2) \cdot (inf(v_1) - inf(v_2)) \\
&< \mathrm{LEP}(t)
\end{aligned}$$

Contradicţie: s-a obţinut un arbore cu lungime externă ponderată mai mică. Rezultă $d_1 \geq d_2$. sfdem

**Lema 9.3.** *Presupunem $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$. Există un arbore în $\mathcal{T}(x)$ cu LEP minimă şi în care vârfurile $x_0$ şi $x_1$ sunt vârfuri frate.*

*Demonstraţie.* Fie $t$ un arbore cu LEP minimă. Fie $v_i$ vârful etichetat cu $x_i$ ($inf(v_i) = x_i$) şi $d_i$ distanţa de la rădăcină la vârful $v_i$, $i = 0, \ldots, n-1$. Deoarece $x_i \leq x_{i+1}$ rezultă, conform lemei 9.2, $d_i \geq d_{i+1}$ (în caz de egalitate $x_i = x_{i+1}$ considerăm pe locul $i$ vârful mai depărtat de rădăcină). Fie $v_i$ vârful frate al vârfului $v_0$. Avem $d_1 \geq d_i$ (deoarece $x_1 \leq x_i$) şi $d_1 \leq d_0 = d_i$ (deoarece $x_1 \geq x_0$ şi $v_0$ şi $v_i$ sunt vârfuri frate) care implică $d_1 = d_i$. În arborele $t$ interschimbăm vârfurile $v_1$ şi $v_i$ şi obţinem un arbore $t'$ care satisface concluzia lemei. sfdem

Lema 9.3 ne conduce direct la ideea algoritmului. Presupunem $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$. Ştim că există un arbore optim $t$ în care $x_0$ şi $x_1$ sunt memorate în vârfuri frate. Tatăl celor două vârfuri va memora $x_0 + x_1$. Ştergând cele două vârfuri ce memorează $x_0$ şi $x_1$ se obţine un arbore $t'$. Fie $t1'$ un arbore optim pentru secvenţa $y = (x_0 + x_1, x_2, \ldots, x_{n-1})$ şi $t1$ arborele obţinut din $t1'$ prin „agăţarea" a două vârfuri cu informaţiile $x_0$ şi $x_1$ de vârful ce memorează $x_0 + x_1$. Avem $\mathrm{LEP}(t1') \leq \mathrm{LEP}(t')$ care implică $\mathrm{LEP}(t1) = \mathrm{LEP}(t1') + x_0 + x_1 \leq \mathrm{LEP}(t') + x_0 + x_1 = \mathrm{LEP}(t)$. Dar cum $t$ este optim, rezultă $\mathrm{LEP}(t1) = \mathrm{LEP}(t)$ şi de aici $t'$ este optim pentru secvenţa $y$. Considerăm în loc de secvenţe de numere secvenţe de arbori şi obţinem algoritmul:

```
procedure lep(x, n)
   1:  B ← {t(x_0), ..., t(x_{n-1})}
   2:  while (#B > 1) do
   3:     alege t1, t2 din B cu inf(rad(t1)), inf(rad(t2)) minime
   4:     construieşte arborele t în care subarborii rădăcinii
   5:       sunt t1, t2 şi inf(rad(t))=inf(rad(t1))+inf(rad(t2))
   6:     B ← (B \ {t1,t2}) ∪ {t}
end
```

În procedura `lep`, $t(x_i)$ desemnează arborele format dintr-un singur vârf etichetat cu $x_i$ iar $rad(t)$ rădăcina arborelui $t$. Iniţial se consideră $n$ arbori cu un singur vârf, care memorează numerele $x_i, i = 0, \ldots, n-1$. Pasul de alegere *greedy* constă în selectarea a doi arbori cu etichetele din rădăcină minime şi construirea unui nou arbore ce va avea rădăcina etichetată cu suma etichetelor din rădăcinile celor doi arbori şi pe cei doi arbori ca subarbori ai rădăcinii (figura 9.2).

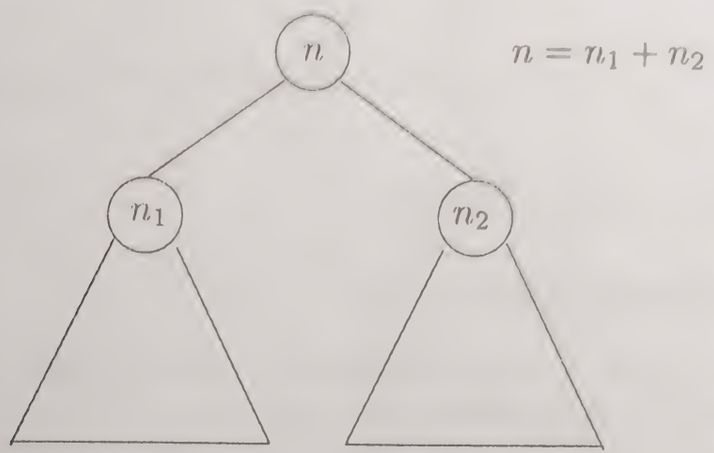



Figura 9.2: Construcţia arborelui din pasul de alegere *greedy*

Discuţia de mai sus conduce la următorul rezultat:

**Teorema 9.2.** *Fie $t^m(x)$ unicul element din mulţimea calculată de schema procedurală* `lep`. *Arborele $t^m(x)$ are proprietatea:*

$$\mathrm{LEP}(t^m(x)) = \min\{\mathrm{LEP}(t) | t \in \mathcal{T}(x)\} \tag{9.2}$$

### 9.3.2 Implementare

Descrierea ca program a algoritmului de mai sus nu are prea mare importanţă practică. De fapt, aşa cum vom vedea şi în secţiunile următoare, soluţia de mai sus constituie mai mult un submodel ce va fi utilizat la rezolvarea unor probleme practice.

Totuşi câteva comentarii sunt necesare. Astfel:

a) dacă mulţimea $B$ este implementată printr-o listă liniară, atunci în cazul cel mai nefavorabil operaţia 3 este are timpul de execuţie $O(n)$, iar operaţia 6 are timpul de execuţie $O(1)$;

b) dacă mulţimea $B$ este implementată printr-o listă liniară ordonată, atunci în cazul cel mai nefavorabil operaţia 3 are timpul de execuţie $O(1)$, iar operaţia 6 are timpul de execuţie $O(n)$;

c) dacă mulţimea $B$ este implementată printr-un *heap*, atunci în cazul cel mai nefavorabil operaţia 3 are timpul de execuţie $O(\log n)$, iar operaţia 6 are timpul de execuţie $O(\log n)$.

Rezultă că pentru implementarea mulţimii $B$, *heapul* este alegerea cea mai bună.

| Literă | Frecvenţă |
|---|---|
| H | 1 |
| A | 4 |
| R | 2 |
| B | 3 |
| U | 1 |

a)

| Literă | Cod |
|---|---|
| H | 010 |
| A | 1 |
| R | 000 |
| B | 001 |
| U | 011 |

b)

Figura 9.3: Codificarea caracterelor din textul HARABABURA

## 9.4 Arbori Huffman

### 9.4.1 Descrierea problemei

$N$ mesaje $M_0, \ldots, M_{n-1}$ sunt recepţionate cu frecvenţele $f_0, \ldots, f_{n-1}$. Mesajele sunt codificate cu şiruri (cuvinte) construite peste alfabetul $\{0, 1\}$ cu proprietatea că pentru orice $i \neq j$, codul mesajului $M_i$ nu este un prefix al codului lui $M_j$. O astfel de codificare se numeşte *independentă de prefix* („prefix-free"). Notăm cu $d_i$ lungimea codului mesajului $M_i$. *Lungimea medie* a codului este $\sum_{i=0}^{n-1} f_i \cdot d_i$. Problema constă în determinarea unei codificări cu lungimea medie minimă.

### 9.4.2 Modelul matematic

Unei codificări îi putem asocia un arbore binar cu proprietatea că mesajele corespund nodurilor de pe frontieră, iar un cod descrie drumul de la rădăcină la mesajul corespunzător: 0 înseamnă deplasarea la fiul stâng, iar 1 deplasarea la fiul drept. Nodurile de pe frontiera arborelui sunt etichetate cu frecvenţele mesajelor corespunzătoare: drumul de la rădăcină la un nod de pe frontieră descrie exact codul mesajului asociat acestui nod. Acum este uşor de văzut că determinarea unui cod optim coincide cu determinarea unui arbore ponderat pe frontieră optim.

**Exemplu.** Codurile Huffman pot fi utilizate la scrierea comprimată a textelor. Considerăm textul HARABABURA. Mesajele sunt literele din text, iar frecvenţele sunt date de numărul de apariţii ale fiecărei litere în text (a se vedea figura 9.3a).

Construcţia arborelui Huffman este reprezentată în figura 9.4. Codurile obţinute sunt reprezentate în figura 9.3b.

sfex

### 9.4.3 Implementare

Presupunem că intrarea este memorată într-un tabel `T` de structuri cu două câmpuri: `T[i].mes` reprezintând mesajul $i$, iar `T[i].f` frecvenţa mesajului $i$. Pentru implementare recomandăm reprezentarea arborilor prin tablouri. Notăm cu `H` tabloul reprezentând arborele Huffman. Semnificaţia câmpului `H[i].elt` este următoarea: dacă $i$ este nod intern, atunci $H[i].elt$ reprezintă informaţia calculată din nod, iar

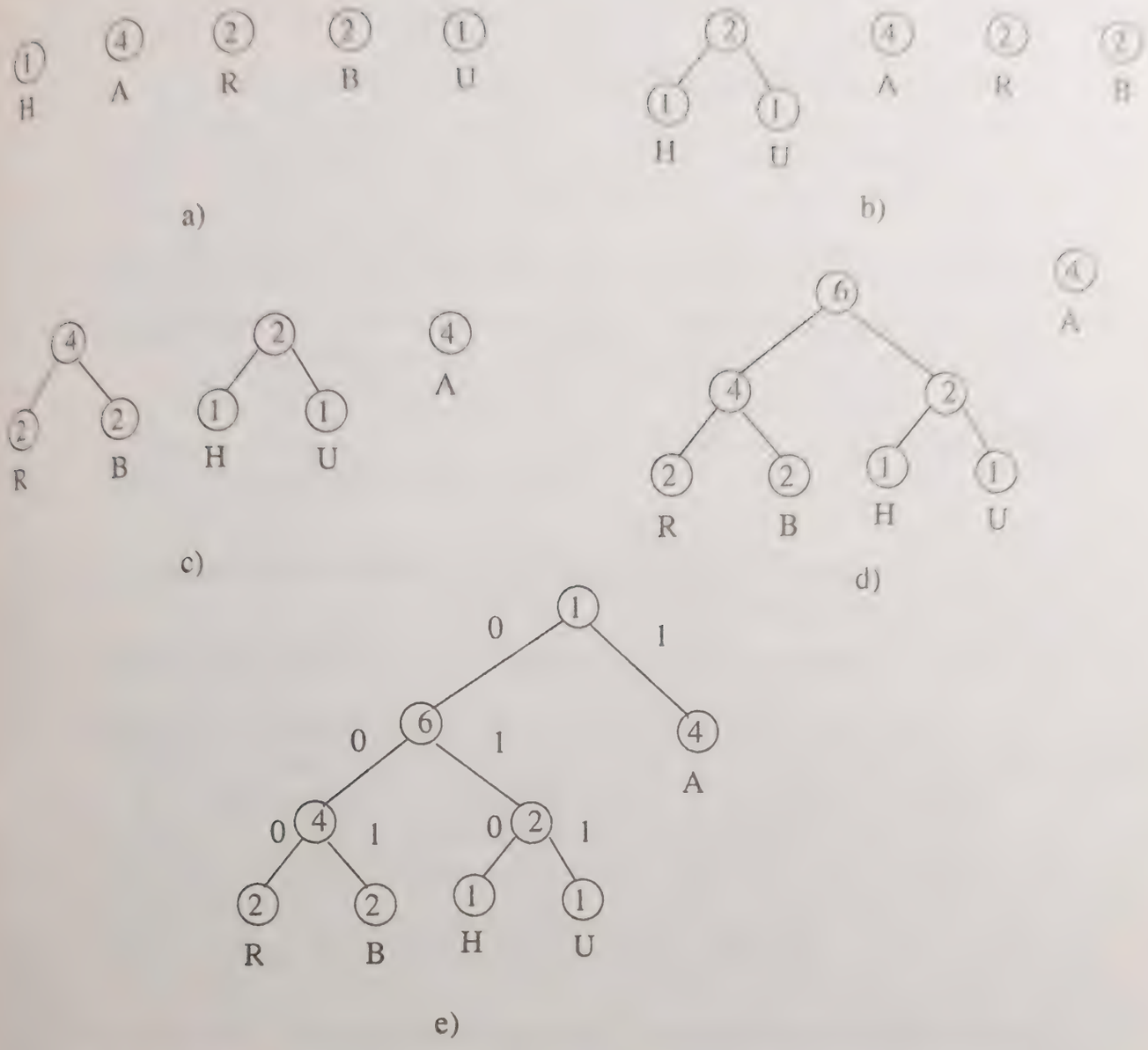



Figura 9.4: Construcţia arborelui Huffman pentru HARABABURA

dacă $i$ este pe frontieră (corespunde unui mesaj), atunci $H[i].elt$ este adresa din T a mesajului corespunzător. În ultimul caz, informaţia din nod este $T[H[i].elt].f$. Notăm cu $val(i)$ funcţia care intoarce informaţia din nodul $i$, calculată ca mai sus. Tabloul H, care în final va memora arborele Huffman corespunzător codurilor optime, va memora pe parcursul construcţiei acestuia colecţiile intermediare de arbori. Astfel că, în timpul execuţiei algoritmului de construcţie a arborelui, H are trei părţi (figura 9.5): prima parte a tabloului va fi un min-*heap* care va conţine rădăcinile arborilor din colecţie. Partea de mijloc va conţine nodurile care nu sunt rădăcini. Cea de-a treia parte este vidă şi constituie zona în care partea din mijloc se poate extinde. Un pas al algoritmului de construcţie ce realizează selecţia *greedy* presupune parcurgerea următoarelor etape:

1. Mută rădăcina cu informaţia cea mai mică pe prima poziţie liberă din zona a treia, să zicem $k$. Aceasta este realizată de următoarele operaţii:
   a) copie rădăcina de pe prima poziţie din heap pe poziţia $k$:
      `H[k] ← H[1]`
      `k ← k + 1`

b) aduce ultimul element din heap pe prima poziţie:

```
H[1] ← H[m]
m ← m - 1
```

c) reface min-*heapul* apelând subprogramul de introducere în min-*heap*.

2. Mută din nou rădăcina cu informaţia cea mai mică pe prima poziţie liberă din zona a treia, dar fără a o elimina din min-*heap*:

```
H[k] ← H[1]
k ← k + 1
```

3. Construieşte noua rădăcină şi o memorează pe prima poziţie în min-*heap* (în locul celei mutate mai sus).
4. Reface min-*heapul* apelând subprogramul de introducere în min-*heap*.

| *heapul* rădăcinilor | noduri care nu nu sunt rădăcini | zonă vidă |
|---|---|---|

Figura 9.5: Organizarea tabloului H

Algoritmul rezultat are timpul de execuţie $O(n \log n)$.

## 9.5 Interclasare optimă pe două căi

### 9.5.1 Descrierea problemei

Se consideră $m$ liste liniare sortate $a_0, \ldots, a_{m-1}$ care conţin $n_0, \ldots, n_{m-1}$, respectiv, elemente dintr-o mulţime total ordonată. Interclasarea celor $m$ secvenţe constă în execuţia repetată a următorului proces: se extrag din mulţime două liste şi se pune în locul lor lista obţinută prin interclasarea acestora. Procesul se continuă până când se obţine o singură listă sortată cu cele $n_0 + \cdots + n_{m-1}$ elemente. Problema constă în determinarea unei alegeri pentru care numărul total de transferuri de elemente să fie minim.

Un exemplu este este dat de *sortarea externă*. Presupunem că avem de sortat un volum mare de date ce nu poate fi încărcat în memoria internă. Soluţia este următoarea: se partiţionează colecţia de date în în mai multe liste ce pot fi ordonate cu unul dintre algoritmii de sortare internă din capitolul 4. Listele sortate sunt memorate în fişiere pe suport extern. Sortarea întregii colecţii se face prin interclasarea fişierelor ce memorează listele sortate.

### 9.5.2 Modelul matematic

Mai întâi considerăm problema interclasării a două liste sortate:

Fie date două liste sortate $x = (x_0, \ldots, x_{m-1})$ şi $y = (y_0, \ldots, y_{n-1})$ ce conţin elemente dintr-o mulţime total ordonată. Să se construiască o listă sortată $z = (z_0, \ldots, z_{n+m-1})$ care să conţină cele $m + n$ elemente ce apar în $x$ şi $y$.

Utilizăm notaţia $z = \text{merge}(x, y)$ pentru a nota faptul că $z$ este rezultatul interclasării listelor $x$ şi $y$. O prezentare a algoritmului de interclasare a două liste ordonate se găseşte în 10.2. Numărul de comparaţii executate de algoritm este cel mult $m + n - 1$, iar numărul de elemente transferate este $m + n$.

Revenim la problema interclasării a $m$ liste. Mai întâi considerăm un exemplu. Fie $m = 5, n_0 = 20, n_1 = 60, n_2 = 70, n_3 = 40, n_4 = 30$. Un mod de alegere a listelor pentru interclasare este următorul:

$$b_0 = \text{merge}(a_0, a_1)$$
$$b_1 = \text{merge}(b_0, a_2)$$
$$b_2 = \text{merge}(a_3, a_4)$$
$$b = \text{merge}(b_1, b_2)$$

Numărul de transferuri al acestei soluţii este $(20 + 60) + (80 + 70) + (40 + 30) + (150 + 70) = 80 + 150 + 70 + 220 = 520$. Există alegeri mai bune? Răspunsul este afirmativ şi invităm cititorul să găsească o astfel de alegere.

Unei alegeri i se poate ataşa un arbore binar în modul următor:

- informaţiile din vârfuri sunt lungimi de secvenţe,
- vârfurile de pe frontieră corespund secvenţelor iniţiale $a_0, \ldots, a_{m-1}$,
- vârfurile interne corespund secvenţelor intermediare.

Pentru soluţia de mai sus avem arborele din figura 9.6a. Se observă uşor că aceştia sunt arbori ponderaţi pe frontieră şi numărul de transferuri de elemente corespunzător unei alegeri este egală cu LEP a arborelui asociat. Aşadar, alegerea optimă corespunde arborelui cu LEP minimă. Algoritmul greedy care interclasează optim cele $m$ liste este următorul:

```
procedure interclOpt(x, n)
begin
   B ← {a_0, ..., a_{n-1}}
   while (#B > 1) do
      alege x1, x2 din B cu lungimi minime
      intercl2(x1, x2, y)
      B ← (B \ {x1,x2}) ∪ {y}
end
```

Pentru exemplul de mai sus, soluţia optimă dată de algoritmul greedy este:

$$b_0 = merge(a_0, a_4)$$
$$b_1 = merge(a_3, b_0)$$
$$b_2 = merge(a_1, a_2)$$
$$b = merge(b_1, b_2)$$

şi are asociat arborele cu lungime externă ponderată minimă din figura 9.6b. Numărul de comparaţii este $50 + 90 + 130 + 220 = 490$.

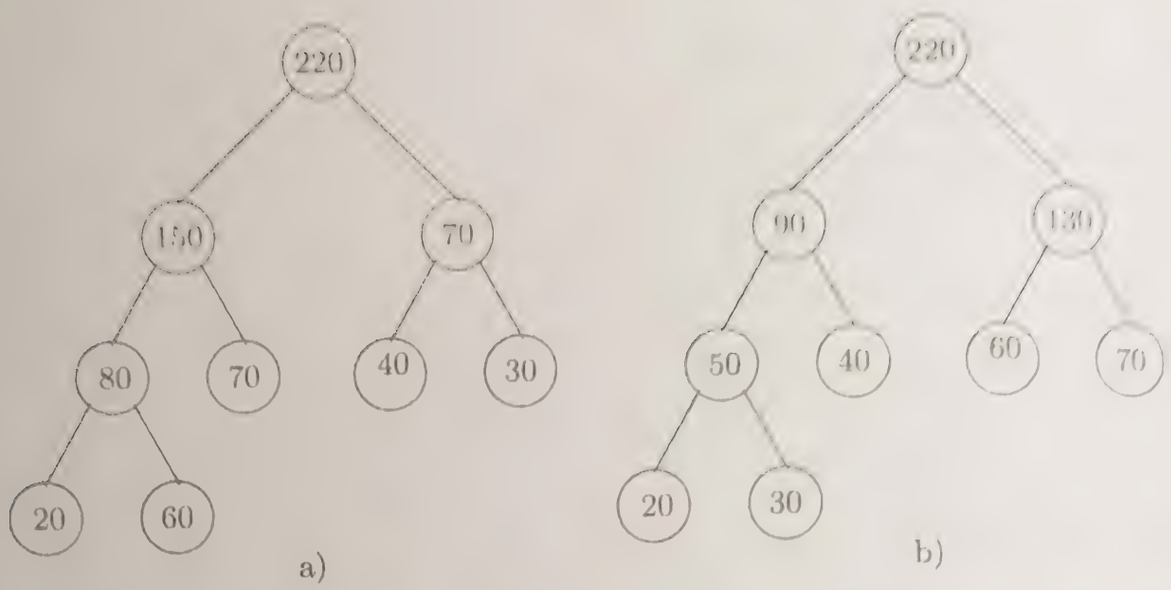



Figura 9.6: Arbori asociaţi algoritmilor de interclasare

## 9.6 Problema rucsacului I (varianta continuă)

### 9.6.1 Descrierea problemei

Se consideră un rucsac de capacitate $M$ şi $n$ obiecte notate cu $0, 1, \ldots, n-1$ de dimensiuni (greutăţi) $w_0, w_1, \ldots, w_{n-1}$. Dacă în rucsac se pune o parte fracţionară $x_i$ din obiectul $i$, $0 \leq x_i \leq 1$, atunci se obţine un profit $p_i \cdot x_i$ ($p_i > 0$). Umplerea rucsacului cu fracţiunile (cantităţile) $x_0, \ldots, x_{n-1}$ aduce profitul total $\sum_{i=0}^{n-1} p_i x_i$. Problema constă în a determina părţile fracţionare $x_0, \ldots x_{n-1}$ care aduc un profit total maxim.

### 9.6.2 Modelul matematic

Problema poate fi formulată ca o problemă de optim, în modul următor:

- funcţia obiectiv:

$$\max \sum_{i=0}^{n-1} p_i x_i$$

- restricţii:

$$\sum_{i=0}^{n-1} w_i x_i \leq M$$
$$0 \leq x_i \leq 1 \text{ pentru } i = 0, \ldots, n-1$$

Dacă $\sum_{i=0}^{n-1} w_i \leq M$, atunci profitul maxim se obţine când $x_i = 1, 0 \leq i \leq n-1$. De aceea vom presupune că $\sum_{i=0}^{n-1} w_i > M$. În acest caz, nu toate fracţiunile $x_i$ pot fi egale cu 1. În plus, rucsacul poate fi umplut exact, *i.e.*, putem alege $x_i$ astfel încât $\sum_{i=0}^{n-1} w_i x_i = M$.

Vom prezenta două strategii greedy: una care nu determină întotdeauna soluţia optimă - pentru a evidenţia că nu întotdeauna alegerea locală cea mai lacomă conduce la soluţia optimă - şi una care dă întotdeauna soluţia optimă.

### 9.6.2.1 Soluţia 1

La fiecare pas se va introduce în rucsac obiectul care aduce profit maxim. La ultimul pas, dacă obiectul nu încape în totalitate, se va introduce numai acea parte fracţionară a sa, care umple exact rucsacul. Descrierea algoritmului este dată de următorul algoritm:

```
procedure rucsac_I1(w, p, x, n)
   S ← {0,...,n − 1}
   for i ← 0 to n-1 do
       x[i] ← 0
       C ← 0
   while ((C < M) and (S ≠ ∅)) do
*:     alege i ∈ S care maximizează profitul peste S
       S ← S \ {i}
       if (C + w[i] ≤ M)
          then C ← C + w[i]
               x[i] ← 1
          else C ← M
               x[i] ← (M − C)/w[i]
end
```

Procedura `rucsac_I1` are dezavantajul că nu determină optimul întodeauna.

**Exemplu.** Presupunem $n = 3, M = 10$, iar dimensiunile şi profiturile obiectelor date de următorul tabel:

| | 0 | 1 | 2 |
|---|---|---|---|
| $w_i$ | 6 | 4 | 8 |
| $p_i$ | 3 | 4 | 6 |

Algoritmul `rucsac_I1` va determina soluţia $x = (0, \frac{1}{2}, 1)$ care produce profitul $\sum p_i x_i = \frac{1}{2} \cdot 4 + 1 \cdot 6 = 8$. Se observă că vectorul $x' = (0, 1, \frac{3}{4})$ produce un profit mai bun: $\sum p_i x'_i = 1 \cdot 4 + \frac{3}{4} \cdot 6 = \frac{17}{2} > 8$. sfex

### 9.6.2.2 Soluţia 2

La fiecare pas va fi introdus în rucsac obiectul care aduce profit maxim pe unitatea de capacitate (greutate) utilizată, adică obiecul care maximizează fracţia $\frac{p_i}{w_i}$ peste mulţimea obiectelor neintroduse încă. Algoritmul corespunzător acestei strategii se obţine din `rucsac_I1` prin înlocuirea liniei * : cu

alege `i ∈ S` care maximizează profitul pe unitatea de greutate peste $S$.

Corectitudinea strategiei este dată de următorul rezultat.

**Teorema 9.3.** *Procedura* `rucsac_I2` *determină soluţia optimă (cu profit maxim).*

*Demonstraţie.* Presupunem $\frac{p_0}{w_0} \geq \cdots \geq \frac{p_{n-1}}{w_{n-1}}$. Fie $x = (x_0, \cdots, x_{n-1})$ soluţia generată de procedura `Rucsac_I2`. Dacă $x_i = 1, 0 \leq i < n$, atunci evident că această

soluţie este optimă. Altfel, fie $j$ primul indice pentru care $x_j \neq 1$. Din algoritm, se observă că $x_i = 1$ pentru orice $0 \leq i < j$ şi $x_i = 0$ pentru $i > j$. Fie $y = (y_0 \cdots y_{n-1})$ o soluţie optimă (care maximizează profitul). Avem $\sum_{i=0}^{n-1} y_i w_i = M$. Fie $k$ cel mai mic indice pentru care $x_k \neq y_k$. Există următoarele posibilităţi:

i) $k < j$. Rezultă $x_k = 1$ şi de aici $y_k \neq x_k$ implică $y_k < x_k$.

ii) $k = j$. Deoarece $\sum x_i \cdot w_i = M$ şi $x_i = y_i, 1 \leq i < j$, rezultă că $y_k < x_k$ (altfel $\sum y_i \cdot w_i > M$ care constituie o contradicţie).

iii) $k > j$. Rezultă $\sum_{i=0}^{n-1} y_i \cdot w_i > \sum_{i=0}^{j} x_i \cdot w_i = M$. Contradicţie.

Deci, toate situaţiile conduc la concluzia $y_k < x_k$ şi $k \leq j$. Mărim $y_k$ cu diferenţa până la $x_k$ şi scoatem această diferenţă din secvenţa $(y_{k+1}, \ldots, y_{n-1})$, astfel încât capacitatea utilizată să rămînă tot $M$. Rezultă o nouă soluţie $z = (z_0, \ldots, z_{n-1})$ care satisface:

$$z_i = x_i, \ 0 \leq i \leq k$$

$$\sum_{k<i\leq n-1} (y_i - z_i) \cdot w_i = (x_k - y_k) \cdot w_k$$

Avem:

$$\begin{aligned}
\sum_{i=0}^{n-1} z_i \cdot p_i &= \sum_{i=0}^{n-1} y_i \cdot p_i + \sum_{0\leq i<k} z_i \cdot p_i + z_k \cdot p_k + \sum_{k<i<n} z_i p_i - \sum_{0\leq i<k} y_i p_i - y_k \cdot p_k - \\
&\quad - \sum_{k<i<n} y_i p_i \\
&= \sum_{i=0}^{n} y_i \cdot p_i + (z_k - y_k) \cdot p_k \cdot \frac{w_k}{w_k} - \sum_{k<i<n} (y_i - z_i) \cdot p_i \cdot \frac{w_i}{w_i} \\
&\geq \sum_{i=0}^{n-1} y_i \cdot p_i + (z_k - y_k) \cdot w_k \frac{p_k}{w_k} - \sum_{k<i<n} (y_i - z_i) \cdot w_i \cdot \frac{p_k}{w_k} \\
&= \sum_{i=0}^{n-1} y_i \cdot p_i
\end{aligned}$$

Deoarece $y$ este soluţie optimă, rezultă $\sum_{i=0}^{n-1} z_i p_i = \sum_{i=0}^{n-1} y_i p_i$. Soluţia $z$ are următoarele două proprietăţi:

- este optimă, şi
- coincide cu $x$ pe primele $k$ poziţii ($y$ coincidea cu $x$ numai pe primele $k-1$ poziţii).

Procedeul de mai sus este repetat (considerând $z$ în loc de $y$) până când se obţine o soluţie optimă care coincide cu $x$. sfdem

**Implementare.** Timpul de execuţie al algoritmului `rucsac_I2` este $O(n^2)$. Dar dacă intrările satisfac $\frac{p_0}{w_0} \geq \cdots \geq \frac{p_{n-1}}{w_{n-1}}$, atunci algoritmul `rucsac_I2` necesită timpul $O(n)$. La acesta trebuie adăugat timpul de preprocesare (ordonare) care este $O(n \log n)$.

## 9.7 Secvenţializarea optimă a activităţilor cu termen de realizare şi profit

### 9.7.1 Descrierea problemei

Considerăm $n$ activităţi numerotate cu $0, 1, \ldots, n-1$. Presupunem că realizarea fiecărei activităţi durează o unitate de timp şi că prima activitate începe la momentul zero. Realizarea activităţii $i$ aduce un profit $p(i) > 0$, $i = 0, \ldots, n-1$. Există constrângerea ca fiecare activitate $i$ să fie terminată la termenul-limită $d(i)$ ($d(i)$ reprezintă durata maximă care este la dispoziţie pentru a executa $i$). Problema constă în determinarea unei liste liniare $(s(0), \ldots, s(k-1))$, care constituie o secvenţializare a realizării activităţilor, astfel încât orice activitate este realizată în termen, *i.e.*, $d(s(i)) \geq i+1$, şi profitul dat de aceasta este maxim.

**Exemplu.** Presupunem $n = 4$. Termenele de execuţie şi profiturile sunt date în următorul tabel:

| $i$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $d(i)$ | 2 | 3 | 1 | 2 |
| $p(i)$ | 20 | 35 | 35 | 25 |

O soluţie este $(0, 3, 1)$ cu profitul $20 + 25 + 35 = 80$, iar o altă soluţie este $(2, 3, 1)$ cu profitul $35 + 25 + 35 = 95$. Există secvenţializări ce aduc un profit mai mare? sfex

### 9.7.2 Modelul matematic

Numim *secvenţă (soluţie) acceptabilă* o listă $(s(0), \ldots, s(k-1))$ cu proprietatea $d(s(i)) \geq i+1, i = 0, \ldots, k-1$, *i.e.*, orice activitate $i$ este realizată în termen. Suntem interesaţi în a găsi o listă liniară $s^*$ cu proprietatea:

$$\sum_i p(s^*(i)) = \max\left\{ \sum_i p(s(i)) \mid s \text{ soluţie acceptabilă} \right\}$$

Numim aceste secvenţe *soluţii optime*. Următorul rezultat arată că, pentru o soluţie acceptabilă, dacă ordonăm activităţile după termenele de predare, atunci se obţine tot o soluţie acceptabilă.

**Teorema 9.4.** *Fie $s = (s(0), \ldots, s(k-1))$ o secvenţializare a $k$ lucrări. Dacă $s$ este acceptabilă şi $\pi = (\pi(0), \ldots, \pi(k-1))$ este o permutare a mulţimii $\{0, \ldots, k-1\}$ cu proprietatea $d(s(\pi(0))) \leq \cdots \leq d(s(\pi(k-1)))$, atunci $s(\pi) = (s(\pi(0)), \ldots, s(\pi(k-1)))$ este o secvenţă acceptabilă.*

*Demonstraţie.* Presupunem că există $i < j$ cu $d(s(i)) > d(s(j))$. Considerăm $s' = (\ldots, s'(i) = s(j), \ldots, s'(j) = s(i), \ldots)$ (în rest coincide cu $s$). Din $d(s(j)) > j$ rezultă $d(s'(i)) > i+1$ şi din $d(s(i)) > d(s(j)) > j$ rezultă $d(s'(j)) > j+1$.

Repetând raţionamentul de mai sus, la un moment dat vom obţine secvenţa care satisface concluzia teoremei. sfdem

**Observaţie.** Teorema de mai sus este adevărată şi în cazul când timpii de execuţie ai activităţilor pot fi diferiţi. sfobs

Teorema 9.4 ne permite să facem următoarea modificare asupra definiţiilor pentru soluţie acceptabilă şi soluţie optimă. Fie $B \subseteq \{0, \ldots, n-1\}$. Spunem că $B$ este o *soluţie acceptabilă* dacă există o secvenţă acceptabilă $(s(0), \ldots, s(k-1))$ cu proprietatea că $B = \{s(0), \ldots, s(k-1)\}$. $B$ este *soluţie optimă* dacă există o secvenţă optimă $s^* = (s^*(0), \ldots, s^*(k-1))$ cu proprietatea că $B = \{s^*(0), \ldots, s^*(k-1)\}$. Cu aceste definiţii putem descrie algoritmul greedy care rezolvă secvenţializarea optimă a activităţilor. Pasul de alegere greedy va selecta de fiecare dată o activitate care aduce un profit maxim şi formează o soluţie acceptabilă cu celelalte activităţi alese până în acel moment:

```
procedure secvact(p, d, n, B)
    X ← {0,...,n − 1}
    B ← ∅
    while (X ≠ ∅) do
        alege i din X cu profit maxim
        X ← X \ {i}
        if B ∪ {i} este soluţie acceptabilă
            then B ← B ∪ {i}
end
```

Corectitudinea algoritmului `secvact` este dată de următoarea teoremă.

**Teorema 9.5.** *Fie $B$ mulţimea determinată de algoritmul* `secvact`. *Orice secvenţializare acceptabilă a lui $B$ este optimă.*

*Demonstraţie.* Presupunem $B = \{i_0, \ldots, i_{k-1}\}$ şi fie $B' = \{j_0, \ldots, j_{k'-1}\}$ o soluţie optimă. Ideea demonstraţiei este următoarea: dacă $B' \neq B$ şi cele două mulţimi au $r$ elemente comune, atunci se construieşte o soluţie optimă $B''$ care are $r+1$ elemente comune cu $B$. Procedeul se repetă până când se obţine o soluţie optimă care coincide cu $B$. Putem presupune, fără să restrângem generalitatea, că $p(i_0) \geq \cdots \geq p(i_{k-1})$ şi că $p(j_0) \geq \cdots \geq p(j_{k'-1})$. În plus, dacă $p(j_a) = p(j_{a+1})$, atunci $j_a < j_{a+1}$. De asemenea, presupunem că algoritmul determină activităţile tot în această ordine (dintre toate activităţile cu acelaşi profit o va alege pe cea cu numărul de ordine indicele – mai mic). Există următoarele posibilităţi:

1. $B \subset B'$. Din modul cum lucrează algoritmul `secvact`, rezultă că $B'$ nu este acceptabilă (la $B$ nu se mai poate adauga nimic astfel încât să se poată obţine o secvenţializare acceptabilă).
2. $B' \subset B$. Rezultă că $B'$ nu este soluţie optimă.
3. $B$ şi $B'$ incomparabile. Vom arăta că putem alege $i_a \in B \setminus B'$ şi $j_b \in B' \setminus B$ cu $d(j_b) \leq d(i_a)$. Alegem $i_a$ cu $a < \min\{k, k'\}$, $i_a \neq j_a$ şi $a$ este cel mai mic cu această proprietate. Fie $T = \{\ell_0, \ldots, \ell_{t-1}\} = \{\ell \neq i_a \mid d(\ell) \leq d(i_a)\}$. Presupunem $d(\ell_0) \leq \cdots \leq d(\ell_{t-1})$. Distingem următoarele două subcazuri:

a) $T \subseteq B$. Rezultă că pentru orice $j_b \in B' \setminus B$ avem $d(j_b) > d(i_a) > t$. Fie $s$ o secvenţializare acceptabilă a lui $B'$ (ordonată crescător după termenele de execuţie). Urmează că secvenţa $(s(0) = \ell_0, \ldots, s(t-1) = \ell_{t-1}, i_a, s(t), \ldots)$ este acceptabilă şi de aici rezultă $B' \cup \{i_a\}$ acceptabilă (în orice secvenţă acceptabilă a lui $B'$ poate fi inserat şi $i_a$). Dar aceasta contrazice faptul că $B'$ este optimă.

b) $\exists j_b \in T$ cu $j_b \notin B$. Din definiţia mulţimii $T$ rezultă $d(i_a) \geq d(j_b)$. Deci $B'' = B' \setminus \{j_b\} \cup \{i_a\}$ este acceptabilă. Din modul de alegere a lui $i_a$ rezultă $p(i_a) \geq p(j_b)$ care implică $B''$ optimă.

Deci în singurul caz posibil am obţinut o soluţie optimă care are $a$ elemente comune cu $B$ ($B'$ avea numai $a-1$ elemente comune cu $B$). Repetând procedeul, vom obţine la un moment dat o soluţie optimă $B^* = B$. Acum teorema este demonstrată complet.

sfdem

### 9.7.3 Implementare

Datorită teoremei 9.4 putem determina direct secvenţa optimă $s^*$ cu $d(s^*(0)) \leq \cdots \leq d(s^*(k-1))$. Vom utiliza un tablou s şi o variabilă k cu proprietatea că $S = \{s[0], \ldots, s[k-1]\}$ şi $d(s[0]) \leq \cdots \leq d(s[k-1])$. Testarea proprietăţii dacă $S \cup \{i\}$ este admisibilă se face prin inserarea lui $i$ în tabloul $s$ astfel încât ordonarea nedescrescătoare a termenilor de execuţie să fie posibilă. Pentru ca algoritmul de inserare să poată fi descris uniform, este util de considerat o activitate $-1$ cu $d(-1) = p(-1) = -1$. Strategia de căutare este următoarea:

- se caută secvenţial indicele $i_0$ cu:

$$d(s[i_0]) \leq d(i) \leq d(s[i_0+1]);$$

- se verifică proprietatea:

$$(\forall j) i_0 + 1 \leq j < k \Rightarrow d(s[j]) \geq j + 1;$$

- dacă rezultatul evaluării este *true*, atunci inserează $i$ la poziţia $i_0 + 1$.

Condiţiile de mai sus pot fi combinate într-una singură:

- se determină secvenţial de la dreapta la stânga primul $i_0$ cu proprietatea:

$$d[s[i_0]] \leq d[i] \vee d[s[i_0]] = i_0$$

- dacă procesul de căutare se termină cu $d[s[i_0]] \leq d[i]$, atunci din $d[s[j]] \geq j \wedge d[s[j]] \neq j$ rezultă $d[s[j]] \geq j+1$, pentru orice $j > i_0$. În acest caz se inserează $i$ pe poziţia $i_0 + 1$.

Alegerea activităţilor de profit maxim este mai uşoară dacă acestea se consideră sortate după profit: $p[0] \geq \cdots \geq p[n-1]$. Descrierea strategiei de mai sus este următoarea:

```
procedure secvact(p, d, n, s, k)
   p[-1] ← 0
   d[-1] ← 0
   s[-1] ← 0
   for i ← 1 to n-1 do
       j ← k
       while ((d[s[j]] > d[i]) and (d[s[j]] > j)) do
          j ← j-1
       if (d[s[j]] ≤ d[i])
          then i0 ← j
               for j ← k downto i0+1 do
                   s[j+1] ← s[j]
               s[i0+1] ← i
               k ← k+1
end
```

O buclă `for` execută $O(k)$ operaţii şi deci timpul de execuţie al algoritmului este $O(1) + \cdots + O(n) = O(n^2)$. Timpul de execuţie poate fi redus aproape de $O(n)$, reprezentând mulţimile prin arbori şi utilizând algoritmii *union* şi *find* pentru operaţiile asupra mulţimilor (secţiunea 2.10).

## 9.8 Problema instructorului de schi

### 9.8.1 Descrierea problemei

Un instructor de schi are la dispoziţie $n$ perechi de schiuri pe care trebuie să le distribuie la $n$ elevi începători. Problema instructorului constă în faptul că distribuirea trebuie făcută în aşa fel încât suma diferenţelor absolute dintre înălţimea elevului şi lungimea schiurilor atribuite să fie minimă.

### 9.8.2 Modelul matematic

Notăm cu $x_0, \ldots, x_{n-1}$ înălţimile elevilor şi cu $y_0, \ldots, y_{n-1}$ lungimile schiurilor. Problema revine la rezolvarea următoarei probleme de optim:

$$\min_{\pi \in \mathbb{S}_n} \sum_{i=0}^{n-1} |x_i - y_{\pi(i)}|,$$

unde prin $\mathbb{S}_n$ am notat mulţimea permutărilor elementelor $\{0, \ldots, n-1\}$. Strategia greedy propusă este următoarea: la cel mai mic elev fără schiuri se atribuie cea mai scurtă pereche de schiuri disponibilă. Descrierea acestei strategii este:

```
procedure ski(x, y, n, B)
   X ← {x0, ..., xn-1}
   Y ← {y0, ..., yn-1}
   B ← ∅
   while ((X ≠ ∅) and (Y ≠ ∅)) do
```

```
    x ← min X
    X ← X \ {x}
    y ← min Y
    Y ← Y \ {y}
    B ← B ∪ {(x, y)}
end
```

Fără să restrângem generalitatea, presupunem $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$. Cu această presupunere, criteriul de alegere locală se reduce la a considera elevii în ordine crescătoare. Corectitudinea strategiei de mai sus este dată de următoarea teoremă:

**Teorema 9.6.** *Notăm cu* $S(\pi)$ *suma* $\sum_{i=0}^{n-1} |x_i - y_{\pi(i)}|$. *Dacă* $y_{\pi(0)} \leq \cdots \leq y_{\pi(n-1)}$, *atunci*

$$S(\pi) = \min\{S(\pi') \mid \pi' \text{ permutare a mulţimii } \{0, 1, \ldots, n-1\}\}.$$

*Demonstraţie.* Fie $\pi$ o permutare oarecare cu $i < j, y_{\pi(i)} > y_{\pi(j)}$. Vom arăta că $S(\pi') \leq S(\pi)$, unde $\pi'$ este produsul lui $\pi$ cu transpoziţia $(i, j)$. Distingem următoarele cazuri:

1. $x_i \leq x_j \leq y_{\pi(i)}$.

   a) $y_{\pi(j)} < x_i \leq x_j \leq y_{\pi(i)}$. Avem:

$$\begin{aligned} |y_{\pi(i)} - x_i| + |x_j - y_{\pi(j)}| &= |y_{\pi(i)} - x_j| + |x_i - y_{\pi(j)}| + 2(x_j - x_i) \\ &\geq |y_{\pi(i)} - x_j| + |x_i - y_{\pi(j)}| \end{aligned}$$

   b) $x_i \leq y_{\pi(j)} \leq x_j \leq y_{\pi(i)}$. Avem:

$$\begin{aligned} |y_{\pi(i)} - x_i| + |x_j - y_{\pi(j)}| &= |y_{\pi(i)} - x_j| + |y_{\pi(j)} - x_i| + 2(x_j - y_{\pi(j)}) \\ &\geq |y_{\pi(i)} - x_j| + |y_{\pi(j)} - x_i| \end{aligned}$$

   c) $x_i \leq x_j \leq y_{\pi(j)} \leq y_{\pi(i)}$. Avem:

$$|y_{\pi(i)} - x_i| + |y_{\pi(j)} - x_j| = |y_{\pi(j)} - x_i| + |y_{\pi(i)} - x_j|$$

2. $y_{\pi(j)} < y_{\pi(i)} < x_j$. Se analizează la fel ca mai sus.

În toate cazurile rezultă:

$$|x_i - y_{\pi(i)}| + |x_j - y_{\pi(j)}| \geq |x_i - y_{\pi(j)}| + |x_j - y_{\pi(i)}|$$

Acum are loc:

$$\begin{aligned} S(\pi') &= \sum_{k \neq i,j} |x_k - y_{\pi(k)}| + |x_i - y_{\pi(j)}| + |x_j - y_{\pi(i)}| \\ &\leq \sum_{k \neq i,j} |x_k - y_{\pi(k)}| + |x_i - y_{\pi(i)}| + |x_j - y_{\pi(j)}| \\ &= S(\pi) \end{aligned}$$

Am obţinut următoarea proprietate:

$$(\forall i,j) i<j,\ y_{\pi(i)} > y_{\pi(j)},\ \pi' = \pi\cdot(i,j) \Rightarrow S(\pi') \le S(\pi)$$

Rezultă că permutarea $\pi$ cu proprietatea $y_{\pi(0)} \le \cdots \le y_{\pi(n-1)}$ realizează minimul funcţiei $S$.

sfdem

**Corolar 9.1.** *Dacă*

$$(\forall i,j) x_i \le y_j \text{ sau } (\forall i,j) y_j \le x_i,$$

*atunci* $S(\pi) = \text{const}$.

**Exerciţiul 9.8.1.** [MS91] Să se arate că algoritmul `ski` minimizează de asemenea şi funcţia $S(\pi) = \max\{|x_i - y_{\pi(i)}| \mid 0 \le i \le n-1\}$ (diferenţa pentru cazul cel mai nefavorabil).

## 9.9 Arborele parţial de cost minim

Următoarea problemă este cunoscută ca *arborele parţial de cost minim*: dat un graf ponderat $G = (V, E)$ cu funcţia de cost $c : E \to \mathcal{R}$, să se determine un arbore parţial de cost minim.

### 9.9.1 Modelul matematic

Reamintim că un arbore parţial al lui $G$ este un subgraf conex fără cicluri ce include toate vârfurile lui $G$. Fie $A$ o mulţime de muchii. O muchie $\{i,j\}$ este *sigură* pentru $A$ dacă $A \cup \{\{i,j\}\}$ este o submulţime a unui arbore parţial de cost minim. Următorul algoritm generic, bazat pe strategia greedy, determină arborele parţial descris ca o mulţime de arce:

```
procedure APCM(V,E,c)
    A ← ∅
    E₁ ← E
    while (V,A) nu formează arbore parţial do
        determină o muchie {i,j} ∈ E₁ sigură A
        E₁ ← E₁ \ {{i,j}}
        A ← A ∪ {{i,j}}
end
```

**Teorema 9.7.** *$(V, A)$ determinat de algoritmul* `APCM` *este arbore parţial de cost minim pentru $G = (V, E)$.*

### 9.9.2 Implementare

Se cunosc doi algoritmi eficienţi bazaţi pe schema de mai sus:

- *Algoritmul lui Kruskal*: mulţimea $A$ este o colecţie de arbori (pădure). Pasul de alegere locală selectează muchia de cost minim care nu formează circuite cu muchiile alese până în acel moment. O implementare eficientă a algoritmului lui Kruskal se obţine reprezentând $A$ printr-o structură *union-find*:

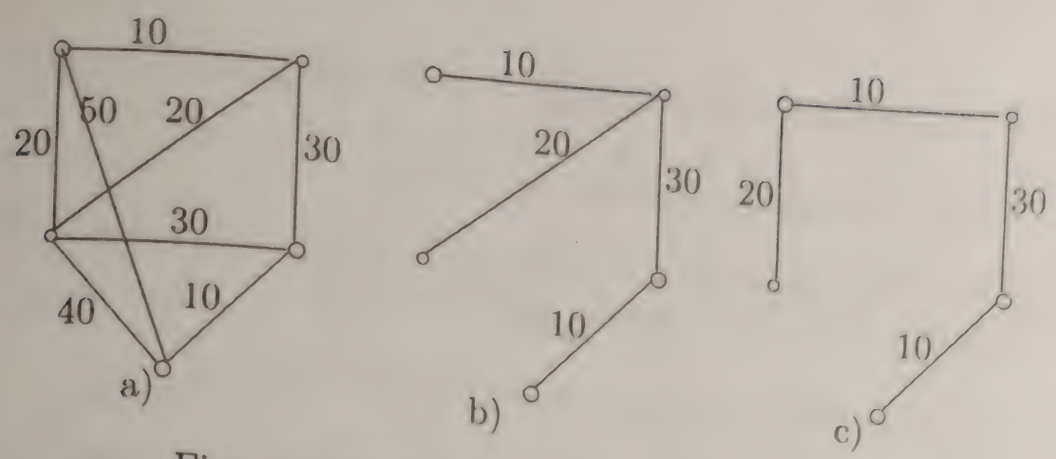



Figura 9.7: Arbore parţial de cost minim

```
procedure Kruskal(V,E,c)
   A ← ∅
   for each i ∈ V do single(i)
   sortează E crescător după c
   for each {i,j} ∈ E în ordine crescătoare do
       if (find(A, i) ≠ find(A, j)
          then union(A,i,j)
end
```

- *Algoritmul lui Prim*: mulţimea $A$ este arbore. Pasul de alegere locală selectează muchia de cost minim care, împreună cu celelalte alese până în acel moment, păstrează proprietatea de arbore. O implementare eficientă a algoritmului lui Prim se obţine prin întreţinerea unei structuri min-*heap*, pe care o notăm cu $Q$. Fiecărui vârf $i$ i se asociază o valoare $cheie(i)$ ce reprezintă minimul dintre costurile muchiilor care unesc acel vârf cu un vârf din arborele construit până în acel moment. Pe timpul execuţiei algoritmului, $Q$ va memora, pe baza valorilor $cheie(i)$ definite mai sus, vârfurile care nu sunt în arbore. Mulţimea $A$ va fi $A = \{\{j, parinte(j)\} \mid j \in V \setminus \{r\} \setminus Q\}$, unde $r$ este rădăcina arborelui (aleasă arbitrar) şi $parinte(j)$ este adresa vârfului care realizează valoarea $cheie(j)$:

```
procedure Prim(V,E,c)
   Q ← V
   for each i ∈ Q do cheie[i] ← ∞
   cheie[r]  ← 0
   parinte[r]  ← -1
   while Q ≠ ∅ do
      if (j ∈ Q and c(i,j) < cheie[j])
         then parinte[j]  ← i
              cheie[j]  ← c({i,j})
end
```

În figura 9.7a este reprezentat un graf poderat $G$, în fig. 9.7b arborele parţial minim determinat cu algoritmul lui Kruskal şi în fig. 9.7c arborele parţial minim determinat cu algoritmul lui Prim (rădăcina este vârful din stânga-sus).

## 9.10 Prezentare formală a paradigmei

### 9.10.1 Modelul matematic

Fie $S$ o mulţime finită de intrări şi $\mathcal{C}$ o colecţie de submulţimi ale lui $S$. Spunem că $\mathcal{C}$ este *accesibilă* dacă satisface *axioma de accesibilitate*:

$$(\forall X \in \mathcal{C}) X \neq \emptyset \Rightarrow (\exists x \in X) X \setminus \{x\} \in \mathcal{C} \tag{9.3}$$

Dacă $\mathcal{C}$ este accesibilă, atunci perechea $(S, \mathcal{C})$ se numeşte *sistem accesibil*. De exemplu, perechea $(S, \mathcal{C})$ definită în secţiunea 9.2 pentru problema numerelor programelor este un sistem accesibil.

**Exerciţiul 9.10.1.** Fie $G = (V, E)$ un graf. Definim $S(G) = E$ şi $\mathcal{C}(G) = \{X \subseteq E \mid (V, X)$ este arbore$\}$. Să se arate că $(S(G), \mathcal{C}(G))$ este sistem accesibil.

**Exerciţiul 9.10.2.** Să se definească un sistem accesibil pentru arborii ponderaţi pe frontieră.

O submulţime $X \in \mathcal{C}$ se numeşte *bază* dacă este maximală, *i.e.* nu există $x \in S \setminus X$ cu $X \cup \{x\} \in \mathcal{C}$. O submulţime $X \in \mathcal{C}$ care nu este bază se numeşte *extensibilă*. Cu alte cuvinte, dacă $X$ este extensibilă, atunci există $y \in S \setminus X$ astfel încât $X \cup \{y\} \in \mathcal{C}$. Clasa de probleme pentru care se pot defini algoritmi greedy este definită de următoarea schemă:

> Se consideră date un sistem accesibil $(S, \mathcal{C})$ şi o *funcţie obiectiv* $f : \mathcal{C} \to \mathbb{R}$. Problema constă în determinarea unei baze $B \in \mathcal{C}$ care satisface:
>
> $$f(B) = \text{optim}\{f(X) \mid X \text{ bază în } \mathcal{C}\}$$

În general, prin optim vom înţelege minim sau maxim. Strategia greedy constă în găsirea unui criteriu de selecţie a elementelor din $S$ care candidează la formarea bazei optime (care dă optimul pentru funcţia obiectiv). Acest criteriu este numit *alegere greedy* sau *alegere a optimului local*. Formal, optimul local are o următoarea definiţie [MS91]:

$$f(X \cup \{x\}) = \text{optim}\{f(X \cup \{y\}) \mid y \in S \setminus X, X \cup \{x\} \in \mathcal{C}\} \tag{9.4}$$

Această strategie este descrisă schematic de următorul algoritm:

```
procedure greedy(S, B)
    S1 ← S
    B ← ∅
    while (B este extensibilă) do
        alege un optim local x din S1 conform cu 9.4
        S1 ← S1 \ {x}
        B ← B ∪ {x}
end
```

Din păcate, numai condiţia de accesibilitate nu asigură existenţa întotdeauna a unui criteriu de alegere locală care să conducă la determinarea unei baze optime. Aceasta înseamnă că, pentru anumite probleme, putem proiecta algoritmi *greedy* care nu furnizează soluţia optimă, ci o bază pentru care funcţia obiectiv poate avea valori apropiate de cea optimă. Acesta este cazul, de exemplu, pentru anumite probleme NP-complete (a se vedea secţiunea 13.3).

#### 9.10.1.1 Matroizi

În această subsecţiune vom prezenta o clasă particulară de probleme pentru care algoritmii *greedy* conduc la determinarea unei baze optime.

Perechea $M = (S, \mathcal{C})$ se numeşte *matroid*, dacă satisface următoarele condiţii:

- *proprietatea de ereditate*:

$$X \in \mathcal{C} \wedge X \neq \emptyset \Rightarrow (\forall x \in X) X \setminus \{x\} \in \mathcal{C}.$$

- *proprietatea de interschimbare*:

$$X, Y \in \mathcal{C} \wedge \#X < \#Y \Rightarrow (\exists y \in Y \setminus X) X \cup \{y\} \in \mathcal{C}.$$

**Exerciţiul 9.10.3.** Să se studieze dacă $(S, \mathcal{C})$ definit în secţiunea 9.2 pentru memorarea programelor este un matroid.

**Exerciţiul 9.10.4.** Fie $G = (V, E)$ un graf. Definim $S(G) = E$ şi $\mathcal{C}(G) = \{X \subseteq E \mid (V, X) \text{ este aciclic}\}$. Să se arate că $(S(G), \mathcal{C}(G))$ este un matroid. Rămâne afirmaţia adevărată dacă în loc de „aciclic" considerăm arbore?

Un *matroid ponderat* este un matroid $M = (S, \mathcal{C})$ în care fiecare element $x \in S$ are asociată o pondere $w(x)$. Problemele pe care le considerăm aici sunt definite după următorul şablon:

Se consideră un matroid ponderat $M = (S, \mathcal{C}, w)$. Dacă $X \subseteq S$, atunci definim $w(X) = \sum_{x \in X} w(x)$. Problema constă în determinarea unei baze $B \in \mathcal{C}$ care satisface:

$$w(B) = \max\{w(X) \mid X \text{ bază în } \mathcal{C}\}$$

Determinarea optimului local constă în selectarea unui $x \in S \setminus X$ cu proprietatea:

$$w(x) = \max\{w(y) \mid y \in S \setminus X, X \cup \{y\} \in \mathcal{C}.\}$$

Deoarece optimul local este determinat numai pe baza ponderilor elementelor rămase, algoritmul *greedy* constă în testarea tuturor elementelor din $S$ în ordinea descrescătoare a ponderilor $w(x)$:

```
procedure greedyMatr(S, w, B)
    sortează S descrescător după w(x)
    B ← ∅
    for each x ∈ S considerat în ordine descrescătoare după w(x) do
        if (B ∪ {x} ∈ C) then B ← B ∪ {x}
end
```

**Teorema 9.8.** *Dacă $M = (S, \mathcal{C}, w)$ este un matroid ponderat, atunci algoritmul* `greedyMatr` *determină o bază optimă.*

*Demonstrație.* Fie $x$ primul element din $S$ ales de algoritmul `greedyMatr`. Mai întâi arătăm următoarea propoziție: există o bază optimă $A$ care conține $x$. Fie $A'$ o bază optimă. Dacă $x \in A'$, atunci luăm $A = A'$. Presupunem că $x \notin A'$. Luăm pentru început $A = \{x\}$. Din definiția algoritmului `greedyMatr`, $A \in \mathcal{C}$. Utilizând proprietatea de interschimbare în mod repetat, adaugăm elemente din $A'$ la $A$ până când $\#A = \#A'$. Există $y \in A'$ cu proprietatea $A = A' \setminus \{x\} \cup \{y\}$. Din modul de alegere a lui $x$, avem $w(x) \geq w(y)$ care implică:

$$w(A) = w(A') - w(x) + w(y) \geq w(A').$$

Deoarece $A'$ este optimă, rezultă $w(A) = w(A')$ și de aici avem că $A$ este optimă. Acum demonstrația propoziției este terminată.

Propoziția de mai sus ne spune că algoritmul *greedy* procedează corect când îl alege pe $x$ la primul pas pentru că acesta aparține unei baze optime. Fie acum $M' = (S', w', \mathcal{C}')$, unde $S' = \{y \in S \mid \{x, y\} \in \mathcal{C}\}$, $w' = w|_{S'}$ (restricția lui $w$ la $S'$) și $\mathcal{C}' = \{X \in \mathcal{C} \mid X \cup \{x\} \in \mathcal{C}\}$. Aplicând proprietatea de ereditate rezultă că $M'$ este matroid și, dacă $A$ este o bază optimă a lui $M$, atunci $A \setminus \{x\}$ este o bază optimă pentru $M'$ (matroizii au proprietatea de substructură optimă). Fie $B$ baza calculată de algoritmul `greedyMatr`. Aplicând un raționament inductiv, obținem că $B \setminus \{x\}$ este bază optimă pentru $M'$. De aici rezultă că $B$ este bază optimă pentru $M$. sfdem

### 9.10.2 Evaluarea timpului de execuție

Presupunem că pasul de alegere *greedy* selectează elemente $x$ în timpul $O(k^p)$ unde $k = \#S_1$ și că testarea condiției $B \cup \{x\} \in \mathcal{C}$ se face în timpul $O(\ell^q)$, unde $\ell = \#B$. Avem $k + \ell \leq n$. Mulțimile $S$ și $B$ pot fi reprezentate astfel încât costul operațiilor $B \cup \{x\}$ și $S1 - \{x\}$ să fie egal cu $O(1)$. Deoarece pasul de alegere este executat de $n$ ori rezultă că strategia are timpul de execuție

$$\begin{aligned} T(n) &= O(n^p + 1^q) + \cdots + O(1^p + n^q) \\ &= O(1^p + \cdots + n^p + 1^q + \cdots + n^q) \\ &= O(n^{p+1} + n^{q+1}) = O(n^{\max(p+1, q+1)}) \end{aligned}$$

*Operațiile peste mulțimile $S_1$ și $B$ vor trebui descrise în funcție de reprezentările concrete ale acestora.*

*În cazul matroizilor ponderați, timpul de execuție pentru cazul cel mai nefavorabil poate fi redusă la $O(n \log n)$, dacă testarea condiției $B \cup \{x\} \in \mathcal{C}$ se face în timpul $O(1)$.*

## 9.11 Exerciții

**Exercițiul 9.1.** [MS91] (*Alocarea optimă a fișierelor pentru rețele de calculatoare*) Se consideră o rețea cu $n$ noduri și o mulțime de fișiere la care au acces toate

nodurile. Se presupune că se cunoaşte în devans o secvenţă de cereri de regăsire/modificare a informaţiei din fişiere. Pentru fiecare cerere se cunoaşte nodul care a iniţiat cererea, fişierul invocat şi numărul de biţi ce urmează a fi transferaţi. O *schemă de alocare* constă într-o atribuire a fiecărui fişier la unul sau mai multe noduri. Având mai multe copii ale aceluiaşi fişier avem un avantaj în regăsirea informaţiei: costul unei regăsiri este zero, dacă fişierul este local, şi este egal cu numărul de biţi transferaţi, dacă fişierul este accesat de la distanţă. Dar multiplicarea fişierelor creşte costul operaţiei de modificare, întrucât fiecare copie trebuie modificată şi astfel numărul de biţi accesaţi pentru modificare este înmulţit cu numărul de copii aflate la distanţă. De asemenea, existenţa mai multor copii duce la creşterea siguranţei în exploatare; dar aceasta se întâmplă până la un punct deoarece este o probabilitate foarte mică să cadă toate nodurile staţiei. Funcţia care dă câştigul în siguranţă depinde de numărul de copii şi se supune următorului principiu: fiecare copie nou adăugată aduce un câştig mai mic decât copia anterioară. Costul unei scheme de alocare se obţine prin însumarea costurilor cererilor de regăsire/modificare din secvenţa dată din care se scade câştigul în siguranţă.

Să se proiecteze un algoritm greedy care să determine o schemă de alocare optimă şi să se demonstreze corectitudinea sa.

**Exerciţiul 9.2.** [HS84] (*Schimbarea banilor*) Fie $A_n = \{a_0, \ldots, a_{n-1}\}$ o mulţime finită de tipuri de monezi. De exemplu: $a_0 = 100$ lei, $a_1 = 50$ lei etc. Presupunem că $a_i$ este întreg pentru orice $i$, $0 \leq i < n$. Pentru fiecare tip există o infinitate de monezi. Se pune problema ca pentru un număr întreg $C$ dat, să se determine numărul minim de monezi a căror sumă este exact $C$ ($C$ poate fi schimbată numai cu monezi de tipuri din $A_n$).

1. Să se arate că dacă $a_0 > \ldots > a_{n-1}$ şi $a_{n-1} \neq 1$, atunci există $C > 0$ pentru care problema nu are soluţie.
2. Să se arate că dacă $a_{n-1} = 1$, atunci problema are întotdeauna soluţie.
3. Să se proiecteze un algoritm greedy pentru cazul particular $A_n = \{k^{n-1}, \ldots, k^0\}$, $k > 1$.
4. Să se arate că algoritmul găsit la 3 nu determină întotdeauna soluţia optimă pentru cazul general.

**Exerciţiul 9.3.** [HS84] (*Memorarea programelor II*) Se consideră $n$ programe de lungimi $\ell_0, \ldots, \ell_{n-1}$ ce urmează a fi memorate pe o bandă. Un program $i$ este regăsit cu frecvenţa $f_i$. Dacă programele sunt memorate în ordinea $\pi(0), \pi(1), \ldots, \pi(n-1)$, atunci timpul mediu de regăsire este:

$$T = \frac{\sum_{j=0}^{n-1} f_{\pi(j)} \cdot \sum_{k=0}^{j} \ell_{\pi(k)}}{\sum_{j=0}^{n-1} f_j}$$

Să se scrie un algoritm greedy care determină ordinea de memorare ce minimizează timpul mediu de regăsire. Să se demonstreze corectitudinea algoritmului.

**Exerciţiul 9.4.** [CLR93] Se consideră o mulţime $S = \{0, 1, 2, \ldots, n-1\}$ de activităţi care utilizează aceeaşi resursă. Fiecare activitate $i$ are un timp de start $s_i$ şi un timp de terminare $t_i$. Dacă o activitate $i$ este selectată, atunci ea se va desfăşura în perioada de timp dată de intervalul semideschis $[s_i, t_i)$. Două activităţi $i$ şi $j$

sunt *compatibile* dacă intervalele $[s_i, t_i)$ şi $[s_j, t_j)$ nu se acoperă, i.e., sau $s_i \geq t_j$ sau $s_j \geq t_i$. Să se proiecteze un algoritm greedy care să determine o mulţime cu număr maxim de activităţi compatibile. Să se dovedească corectitudinea algoritmului.

**Exerciţiul 9.5.** Următorul algoritm, cunoscut sub numele de algoritmul Dijkstra, determină drumurile minime într-un digraf ponderat $D = (\langle V, A\rangle, \ell)$ care pleacă dintr-un vârf $i_0$ dat, în cazul când ponderile $\ell[i,j]$ sunt $\geq 0$. Pentru fiecare vârf $i$, $d[i]$ va fi lungimea drumului minim de la $i_0$ la $i$ şi $p[i]$ va fi predecesorul lui $i$ pe drumul minim de la $i_0$ la $i$. $Q$ este un min-*heap* cu cheile date de valorile $d[i]$.

```
procedure Dijkstra(D, i0, d, p)
begin
   for i ← 1 to n do
       p[i] ← 0
       d[i] ← ∞
    d[i0] ← 0
    Q ← V
    while Q ≠ ∅ do
       i ← citeste(Q)
       elimina(Q)
       for fiecare j ∈ listaDeAdiacenta(i) do
           if (d[j] > d[i]+ℓ[i,j])
           then d[j] ← d[i] + ℓ[i,j]
                p[j] ← i
end
```

Se cere:

1. Să se exemplifice execuţia algoritmului Dijkstra pentru digraful din figura 9.8.
2. Notăm cu $\delta(i_0, i)$ lungimea drumului minim de la $i_0$ la $i$ (când există) şi cu $S(t)$ mulţimea vârfurilor $i$ eliminate din Q până la momentul $t$. Să se arate că dacă $i \in S(t)$, atunci $d[i] = \delta[i_0, i]$.
3. Să se arate că algoritmul Dijkstra determină corect drumurile minime care pleacă din $i_0$ (adică $d[i] = \delta[i_0, i]$ pentru orice $i$, după terminare).
4. Să se determine complexitatea algoritmului Dijkstra.
5. Să se arate că algoritmul Dijkstra este un algoritm greedy.

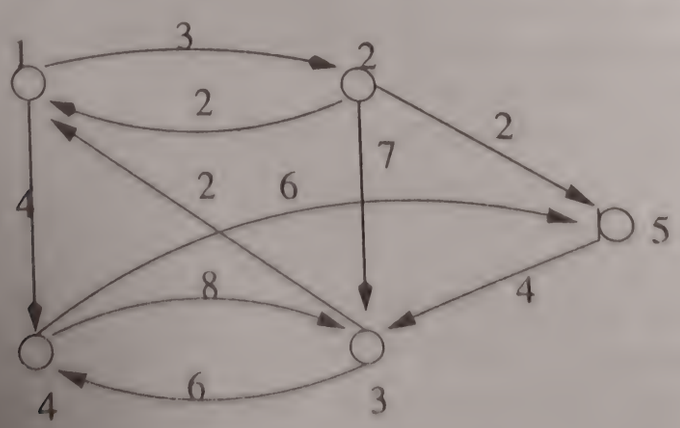



Figura 9.8: Exerciţiul 9.5

## 9.12 Referinţe bibliografice

Prezentarea din acest capitol se bazează pe [HS84, MS91, CLR93, CLR00, Sah98, Hei96, Sed88, LG86]. Pentru aspectele teoretice privind metoda greedy recomandăm [MS91, Cro92, CLR93].

# Capitolul 10

# *Divide-et-impera*

## 10.1 Prezentare generală

### 10.1.1 Modelul matematic

Paradigma *divide-et-impera* (în engleză *divide-and-conquer*) constă în divizarea problemei iniţiale în două sau mai multe subprobleme de dimensiuni mai mici, apoi rezolvarea în aceeaşi manieră (recursivă) a subproblemelor şi combinarea soluţiilor acestora pentru a obţine soluţia problemei iniţiale. Divizarea unei probleme se face până când se obţin subprobleme de dimensiuni mici ce pot fi rezolvate prin tehnici elementare. Paradigma poate fi descrisă schematic astfel: x

```
procedure divideEtImpera(P, n, S)
   if (n ≤ n0)
      then rezolvă subproblema P prin tehnici elementare
      else împarte P in P1, ..., Pa de dimensiuni n1, ..., na
           divideEtImpera(P1, n1, S1)
           ...
           divideEtImpera(Pa, na, Sa)
           combină S1, ..., Sa pentru a obţine S
end
```

Exemple tipice de aplicare a paradigmei *divide-et-impera* sunt algoritmii de parcurgere a arborilor binari şi algoritmul de căutare binară în mulţimi total ordonate. Deoarece descrierea strategiei are un caracter recursiv, aplicarea ei trebuie precedată de o generalizare de tipul problemă $\mapsto$ subproblemă prin care dimensiunea problemei devine o variabilă liberă.

Vom presupune că dimensiunea $n_i$ a subproblemei $P_i$ satisface $n_i \leq \frac{n}{b}$, unde $b > 1$. În acest fel pasul de divizare reduce o subproblemă la altele de dimensiuni mai mici, ceea ce asigură proprietatea de terminare a subprogramului recursiv. De asemenea, descrierea recursivă ne va permite utilizarea inducţiei pentru demonstrarea corectitudinii.

### 10.1.2 Evaluarea timpului de execuţie

Presupunem că divizarea problemei în subprobleme şi asamblarea soluţiilor necesită timpul $O(n^k)$. Timpul de execuţie $T(n)$ a algoritmului `divideEtImpera` este dat de următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} O(1) & \text{, dacă } n \leq n_0 \\ a \cdot T(\frac{n}{b}) + O(n^k) & \text{, dacă } n > n_0 \end{cases} \tag{10.1}$$

**Teorema 10.1.** *Dacă $n > n_0$ atunci:*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{, dacă } a > b^k \\ O(n^k \log_b n) & \text{, dacă } a = b^k \\ O(n^k) & \text{, dacă } a < b^k \end{cases} \tag{10.2}$$

*Demonstraţie.* Fără să restrângem generalitatea presupunem $n = b^m \cdot n_0$. De asemenea mai presupunem că $T(n) = cn_0^k$ dacă $n \leq n_0$ şi $T(n) = aT(\frac{n}{b}) + cn^k$ dacă $n > n_0$. Pentru $n > n_0$ avem:

$$\begin{aligned}
T(n) &= aT(\tfrac{n}{b}) + cn^k \\
&= aT(b^{m-1}n_0) + cn^k \\
&= a(aT(b^{m-2}n_0) + c\left(\frac{n}{b}\right)^k) + cn^k \\
&= a^2T(b^{m-2}n_0) + c\left[a(\tfrac{n}{b})^k + n^k\right] \\
&= \cdots \\
&= a^mT(n_0) + c\left[a^{m-1}\left(\frac{n}{b^{m-1}}\right)^k + \cdots + a\left(\frac{n}{b}\right)^k + n^k\right] \\
&= a^mcn_0^k + c\left[a^{m-1}b^kn_0^k + \cdots + a\left(b^{m-1}\right)^k n_0^k + (b^m)^k n_0^k\right] \\
&= cn_0^k a^m \left[1 + \frac{b^k}{a} + \cdots + \left(\frac{b^k}{a}\right)^m\right] \\
&= ca^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i
\end{aligned}$$

unde am renotat $cn_0^k$ prin $c$. Distingem cazurile:

1. $a > b^k$. Seria $\sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$ este convergentă şi deci şirul sumelor parţiale este convergent. De aici rezultă $T(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a})$.
2. $a = b^k$. Rezultă $a^m = b^{km} = cn^k$ şi de aici $T(n) = O(n^k m) = O(n^k \log_b n)$.
3. $a < b^k$. Avem $T(n) = O(a^m(\frac{b^k}{a})^m) = O(b^{km}) = O(n^k)$.

Acum teorema este demonstrată complet. sfdem

## 10.2 Sortare prin interclasare (*Merge Sort*)

### 10.2.1 Modelul matematic

Considerăm cazul când lista liniară ce urmează a fi sortată este reprezentată ca tablou. Ştim că prin interclasarea a două liste sortate obţinem o listă sortată ce conţine toate elementele listelor de intrare. Ideea este de a utiliza interclasarea în etapa de asamblare a soluţiilor: în urma rezolvării recursive a subproblemelor rezultă liste ordonate şi prin interclasarea lor obţinem lista iniţială sortată. Primul pas constă în generalizarea problemei. Presupunem că se cere sortarea unui tablou `a[p..q]` cu `p` şi `q` variabile libere, în loc de sortarea tabloului `a[0..n-1]` cu `n` variabilă legată (deoarece este dată de intrare). Divizarea problemei constă în împărţirea listei de sortat în două subliste $a[p..m]$ şi $a[m+1..q]$, de preferat de lungimi aproximativ egale. Noi vom lua $m = \left[\frac{p+q}{2}\right]$. Aşa cum am spus, faza de combinare a soluţiilor constă în interclasarea celor două subliste, după ce ele au fost sortate recursiv prin acelaşi procedeu. Pasul de bază este dat de faptul că sublistele de lungime 1 sunt ordonate prin definiţie:

```
procedure mergeSort(a, p, q)
   if (p < q)
      then m ← [(p+q)/2]
           mergeSort(a, p, m)
           mergeSort(a, m+1, q)
           interclasează subtablourile (a[p],...,a[m]), (a[m+1],...,a[q])
                utilizând un tablou temporar
end
```

Pasul de divizare a problemei în subprobleme se face în timpul $O(1)$. Pentru faza de asamblare a soluţiilor, reamintim că interclasarea a două secvenţe ordonate crescător se face în timpul $O(m_1 + m_2)$, unde $m_1$ şi $m_2$ sunt lungimile celor două secvenţe. Aplicând teorema 10.1 pentru $a = 2, b = 2, k = 1$ rezultă că algoritmul `mergeSort` va efectua sortarea unui tablou de lungime $n$ în timpul $O(n \log_2 n)$.

### 10.2.2 Implementare

Listele parţiale obţinute în timpul interclasării vor fi memorate temporar într-o variabilă tablou auxiliară. După terminarea procesului de interclasare, lista rezultat va fi copiată în locul subsecvenţelor de intrare. Astfel programul va utiliza $O(n + \log_2 n)$ memorie suplimentară (tabloul auxiliar şi stiva cu apelurile recursive).

```
procedure intercl2(a, p, q, m, temp)
   i ← p
   j ← m+1
   k ← -1
   while ((i ≤ m) and (j ≤ q)) do
      k ← k+1
      if (a[i] ≤ a[j])
         then temp[k] ← a[i]
```

```
                i ← i+1
            else temp[k] ← a[j]
                j ← j+1
        while (i ≤ m) do
            k ← k+1
            temp[k] ← a[i]
            i ← i+1
        while (j ≤ n) do
            k ← k+1
            temp[k] ← a[j]
            j ← j+1
    end

    procedure mergeSort(a, p, q)
        if (p < q)
            then m ← ⌊(p+q)/2⌋
                 mergeSort(a, p, m)
                 mergeSort(a, m+1, q)
                 intercl2(a, p, q, m, temp)
                 for i ← p to q do
                     a[i] ← temp[i-p]
    end
```

**Exerciţiul 10.2.1.** Să se rescrie `mergeSort` pentru cazul când lista de sortat este reprezentată printr-o listă liniară simplu înlănţuită. Să se compare eficienţa noului algoritm cu cel corespunzător reprezentării cu tablouri.

### 10.2.2.1 Variantă nerecursivă

În continuare vom căuta să găsim o variantă nerecursivă pentru `mergeSort`. Pentru aceasta să observăm că arborele subproblemelor generat de metodă (echivalent arborele apelurilor recursive) este un arbore binar în care vârfurile de pe frontieră corespund subsecvenţelor de lungime 1. Varianta nerecursivă va proceda la parcugerea acestui arbore în maniera "bottom-up", *i.e.*, parcurge nivel cu nivel de la frontieră spre rădăcină: întâi sunt vizitate toate vârfurile de pe nivelul cel mai mare (cel imediat superior frontierei), apoi după vizitarea vârfurilor de pe nivelul $h$ se trece la vizitarea vârfurilor de pe nivelul $h-1$. Vizitarea unui vârf al arborelui constă de fapt în interclasarea a două subtablouri. Deoarece adresa unui vârf poate fi calculată printr-o relaţie simplă, nu este nevoie de gestionarea unei cozi care să memoreze adresele vârfurilor ce urmează a fi vizitate. De remarcat că nivelul vizitat curent în arbore dă şi lungimea subsecvenţelor sortate deja. Frontiera corespunde subsecvenţelor de lungime 1, nivelul imediat superior subsecvenţelor de lungime 2, următorul subsecvenţelor de lungime 4 şi aşa mai departe (figura 10.1).

Pentru a lucra corect, utilizăm un tablou temporar `temp` cu următorul scop: dacă se interclasează subsecvenţe din secvenţa `a`, atunci rezultatul interclasării va fi memorat în `temp`, iar dacă se interclasează subsecvenţe din `temp`, atunci rezultatul

va fi memorat în a. În acest fel, listele intermediare (sortate parţial) vor fi memorate alternativ în cele două tablouri a şi temp.

Faza de interclasare a subsecvenţelor de pe un nivel este realizată în modul următor: se interclasează primele două subsecvenţe, apoi se interclasează a treia subsecvenţă cu a patra ş.a.m.d. S-a preferat o descriere mai condensată a metodei de interclasare.

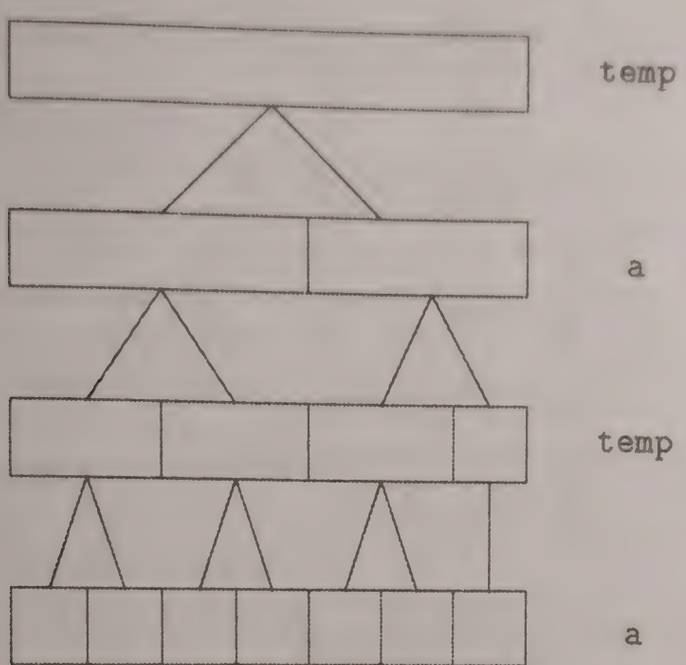



Figura 10.1: *Merge Sort* nerecursiv

```
procedure intercl(secvin, secvout, n, ls)
   i ← 0
   j ← ls
   k ← 0
   repeat
      t ← j
      u ← j+ls
      if (u > n) then u ← n
      while ((i < t) and (j < u)) do
         if (secvin[i] ≤ secvin[j])
            then secvout[k] ← secvin[i]
                 i ← i+1
                 k ← k+1
            else secvout[k] ← secvin[j]
                 j ← j+1
                 k ← k+1
      while (i < t) do
         secvout[k] ← secvin[i]
         i ← i+1
         k ← k+1
      while (j < u) do
```

```
            secvout[k] ← secvin[j]
            j ← j+1
            k ← k+1
        i ← u
        j ← i+ls
    until (j > n-1)
    for j ← i to n-1 do
        secvout[j] ← secvin[j]
end

procedure mergeSortNerec(a, p, q)
    ina ← true
    ls ← 1
    while (ls < n) do
        if (ina)
            then intercl(a,temp,n,ls)
            else intercl(temp,a,n,ls)
        ina ← not ina
        ls ← ls*2
    if (not ina)
        then for i ← 0 to n-1 do
                a[i] ← temp[i]
end
```

**Exerciţiul 10.2.2.** Memorarea alternativă a listelor intermediare în cele două tablouri a şi b reduce considerabil numărul transferurilor. Se poate aplica aceeaşi tehnică şi în cazul subprogramului recursiv `mergeSort`? Dacă răspunsul este da, să se arate cum, iar dacă este nu, să se dea o justificare a acestuia.

**Exerciţiul 10.2.3.** Să se rescrie `mergeSortNerec` pentru cazul când lista de sortat este reprezentată printr-o listă liniară simplu înlănţuită. Să se evidenţieze avantaje/dezavantaje ale utilizării acestei reprezentări.

## 10.3 Sortarea rapidă (*Quick Sort*)

### 10.3.1 Modelul matematic

Ca şi în cazul algoritmului *Merge Sort*, vom presupune că trebuie sortat tabloul `a[p..q]`. Divizarea problemei constă în alegerea unei valori $x$ din $a[p..q]$ şi determinarea prin interschimbări a unui indice $k$ cu proprietăţile:

- $p \leq k \leq q$ şi $a[k] = x$;
- $\forall i : p \leq i \leq k \Rightarrow a[i] \leq a[k]$;
- $\forall j : k < j \leq q \Rightarrow a[k] \leq a[j]$;

Elementul $x$ este numit *pivot*. În general, se alege pivotul $x = a[p]$, dar nu este obligatoriu. Partiţionarea tabloului se face prin interschimbări care menţin invariante proprietăţi asemănătoare cu cele de mai sus. Se consideră două variabile index: `i` cu care se parcurge tabloul de la stânga la dreapta şi `j` cu care se parcurge

tabloul de la dreapta la stânga. Inițial se ia $i = p + 1$ și $j = q$. Proprietățile menținute invariante în timpul procesului de partiționare sunt:

$$\forall i' : p \leq i' < i \Rightarrow a[i'] \leq x \tag{10.3}$$

și

$$\forall j' : j < j' \leq q \Rightarrow a[j'] \geq x \tag{10.4}$$

Presupunem că la momentul curent sunt interogate elementele a[i] și a[j] cu $i < j$.

Distingem următoarele cazuri:

1. $a[i] \leq x$. Transformarea i $\leftarrow i + 1$ păstrează 10.3.
2. $a[j] \geq x$. Transformarea j $\leftarrow j - 1$ păstrează 10.4.
3. $a[i] > x > a[j]$. Dacă se realizează interschimbarea a[i] $\leftrightarrow$ a[j] și se face i $\leftarrow i + 1$ și j $\leftarrow j - 1$, atunci ambele predicate 10.3 și 10.4 sunt păstrate.

Operațiile de mai sus sunt repetate până când $i$ devine mai mare decât $j$. Considerând $k = i - 1$ și interschimbând a[p] cu a[k] obținem partiționarea dorită a tabloului.

După sortarea recursivă a subtablourilor a[p..k − 1] și a[k + 1..q] se observă că tabloul este sortat deja. Astfel partea de asamblare a soluțiilor este vidă.

Algoritmul rezultat este:

```
procedure quickSort(a, p, q)
   if (p < q)
      then determină prin interschimbări indicele k cu:
                  p ≤ k ≤ q
                  (∀)i : p ≤ i ≤ k ⇒ a[i] ≤ a[k]
                  (∀)j : k < j ≤ q ⇒ a[k] ≥ a[j]
           quickSort(a, p, m)
           quickSort(a, m+1, q)
end
```

Cazul cel mai nefavorabil se obține atunci când la fiecare partiționare se obține una din subprobleme cu dimensiunea 1. Deoarece operația de partiționare necesită $O(q-p)$ comparații, rezultă că pentru acest caz numărul de comparații este $O(n^2)$. Acest rezultat este oarecum surprinzător, având în vedere că numele metodei este „sortare rapidă". Așa cum vom vedea, într-o distribuție normală, cazurile pentru care QuickSort execută $n^2$ comparații sunt rare, fapt care conduce la o complexitate medie foarte bună a algoritmului. În continuare determinăm numărul mediu de comparații. Presupunem că $q + 1 - p = n$ (lungimea secvenței) și că probabilitatea ca pivotul $x$ să fie al $k$-lea element este $\frac{1}{n}$ (fiecare element al tabloului poate fi pivot cu aceeași probabilitate $\frac{1}{n}$). Rezultă că probabilitatea obținerii subproblemelor de dimensiuni $k - p = i - 1$ și $q - k = n - i$ este $\frac{1}{n}$. În procesul de partiționare, un element al tabloului (pivotul) este comparat cu toate celelalte, astfel că sunt necesare $n - 1$ comparații. Acum numărul mediu de comparații se calculează prin formula:

$$T^{med}(n) = \begin{cases} (n-1) + \frac{1}{n}\sum_{i=1}^{n}(T^{med}(i-1) + T^{med}(n-i)) & \text{, dacă } n \geq 1 \\ 1 & \text{, dacă } n = 0 \end{cases}$$

Rezolvăm această recurenţă. Avem:

$$T^{med}(n) = (n-1) + \frac{2}{n}(T^{med}(0) + \cdots + T^{med}(n-1))$$
$$nT^{med}(n) = n(n-1) + 2(T^{med}(0) + \cdots + T^{med}(n-1))$$

Trecem pe $n$ în $n-1$:

$$(n-1)T^{med}(n-1) = (n-1)(n-2) + 2(T^{med}(0) + \cdots + T^{med}(n-2))$$

Scădem:

$$nT^{med}(n) = 2(n-1) + (n+1)T^{med}(n-1)$$

Împărţim prin $n(n+1)$ şi rezolvăm recurenţa obţinută:

$$\begin{aligned}
\frac{T^{med}(n)}{n+1} &= \frac{T^{med}(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\
&= \frac{T^{med}(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} - \left(\frac{2}{(n-1)n} + \frac{2}{n(n+1)}\right) \\
&= \cdots \\
&= \frac{T^{med}(0)}{1} + \frac{2}{1} + \cdots + \frac{2}{n+1} - \left(\frac{2}{1\cdot 2} + \cdots + \frac{2}{n(n+1)}\right) \\
&= 1 + 2\left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n+1}\right) - 2\left(\frac{1}{1\cdot 2} + \cdots + \frac{1}{n(n+1)}\right)
\end{aligned}$$

Deoarece $1 + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log_2 n)$ şi seria $\sum \frac{1}{k(k+1)}$ este convergentă (şi deci şirul sumelor parţiale este mărginit), rezultă că $T(n) = O(n \log_2 n)$.

Am demonstrat următorul rezultat:

**Teorema 10.2.** *Complexitatea medie a algoritmului QuickSort este* $O(n \log_2 n)$.

### 10.3.2 Implementare

Subprogramul `partitioneaza` descrie algoritmul de divizare de mai sus:

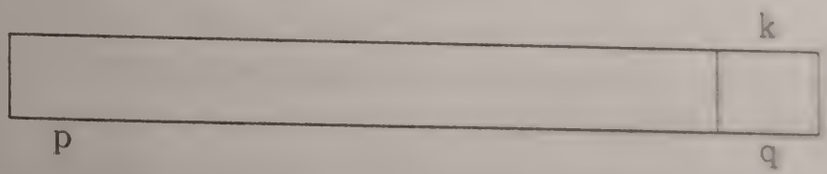
```
procedure partitioneaza(a, p, q, k)
   x ← a[p]
   i ← p + 1
   j ← q
   while (i ≤ j) do
      if (a[i] ≤ x then i ← i + 1
      if (a[j] ≥ x then j ← j - 1
      if (i < j)
         then if ((a[i] > x) and (x > a[j]))
                 then swap(a[i], a[j])
              i ← i + 1
              j ← j - 1
   k ← i-1
   a[p] ← a[k]
   a[k] ← x
end
```

Dacă se presupune că are loc $a[p-1] \leq a[i] \leq a[q+1]$, pentru orice $i$ cu $p \leq i \leq q$ (aceasta implică existenţa iniţială a două elemente artificiale $a[-1]$ şi $a[n]$ cu $a[-1] \leq a[i] \leq a[n]$, $0 \leq i \leq n-1$), atunci procedura de partiţionare poate fi descrisă mai simplu astfel:

```
procedure partitioneaza(a, p, q, k)
   x ← a[p]
   i ← p - 1
   j ← q + 1
   while (i < j) do
      repeat i ← i + 1 until (a[i] ≥ x)
      repeat j ← j - 1 until (a[j] ≤ x)
      if (i < j)
         then swap(a[i], a[j])
   k ← j
   a[p] ← a[k]
   a[k] ← x
end
```

### 10.3.3 Spaţiul ocupat de algoritmul QuickSort

La execuţia algoritmilor recursivi, importantă este şi spaţiul de memorie ocupat de stivă (a se vedea secţiunea 3). Considerăm spaţiul de memorie ocupat de stivă în cazul cel mai defavorabil, $k = q$:




În acest caz, spaţiul de memorie ocupat de stivă este $M(n) = c + M(n-1)$, care implică $M(n) = O(n)$.

În general, pivotul împarte secvenţa de sortat în două subsecvenţe. Dacă subsecvenţa mică este rezolvată recursiv, iar subsecvenţa mare este rezolvată iterativ, atunci consumul de memorie se reduce. Noul algoritm are următoarea descriere

```
procedure QuickSort(a, p, q)
   while (p < q) do
      partitioneaza(a, p, q, k)
      if (k-p > q-k)
         then QuickSort(a, k+1, q)
              q ← k-1
         else QuickSort(a, p, k-1)
              p ← k+1
end
```

Spaţiul de memorie ocupat de stivă pentru noul algoritm satisface relaţia $M(n) \leq c + M(n/2)$, de unde rezultă $M(n) = O(\log n)$.

**Exerciţiul 10.3.1.** Dacă tabloul a conţine multe elemente egale, atunci algoritmul `partitioneaza` realizează multe interschimbări inutile (de elemente egale). Să se modifice algoritmul astfel încât noua versiune să elimine acest inconvenient.

**Exerciţiul 10.3.2.** Să se modifice subprogramul de partiţionare de mai sus prin înlocuirea instrucţiunii `while` cu `repeat` şi a celor două instrucţiuni `repeat` cu `while`. Atenţie: partea de iniţializare şi partea de interschimbare vor fi modificate corespunzător pentru ca noul subprogram să rezolve corect problema partiţionării.

**Exerciţiul 10.3.3.** Următorul program poate fi de asemenea utilizat cu succes pentru partiţionarea tabloului `a[p..q]`:

```
procedure partitioneaza(a, p, q, k)
   i ← p
   j ← q
   iinc ← 0
   jinc ← -1
   while (i < j) do
      if (a[i] > a[j])
         then swap(a[i], a[j])
              incaux ← iinc
              iinc ← -jinc
              jinc ← -incaux
      i ← i+iinc
      j ← j+jinc
   k ← i
end
```

Să se demonstreze că programul face o partiţionare corectă. Ce proprietăţi invariante păstrează instrucţiunea `while`?

Acum, programul care descrie algoritmul QuickSort este următorul:

```
procedure quickSort(a, p, q)
   if (p < q)
      then partitioneaza(a,p,q,k)
           quickSort(a, p, k)
           quickSort(a, k+1, q)
end
```

## 10.4 Selecţionare

Următoarea problemă este o variantă a celei din exerciţiul 4.1.8:

> Să se scrie un algoritm care având la intrare un tablou $(a[i] \mid 0 \leq i < n)$ şi un număr $k \in \{0, 1, \ldots, n-1\}$, schimbă ordinea elementelor din $a$ astfel încât la ieşire pe poziţia $k$ se află cel de-al $k+1$-lea număr cel mai mic (echivalent, cel de-al $n-1-k$-lea cel mai mare), elementele aflate la stânga lui $k$ sunt mai mici decât $a[k]$ şi elementele aflate la dreapta lui $k$ sunt mai mari decât $a[k]$.

Evident, orice algoritm de sortare rezolvă problema de mai sus. Deoarece cerinţele pentru selecţie sunt mai slabe decât cele de la ordonare, se pune firesc întrebarea dacă există algoritmi mai performanţi decât cei utilizaţi la sortare. Condiţia pe care trebuie să o satisfacă la ieşire tabloul a este formulată de:

$$(\forall i)(i < k \Rightarrow a[i] \leq a[k]) \wedge (i > k \Rightarrow a[i] \geq a[k])$$

Fie $j$ o poziţie cu proprietatea:

$$(\forall i)(i < j \Rightarrow a[i] \leq a[j]) \wedge (i > j \Rightarrow a[i] \geq a[j])$$

Un asemenea $j$ poate fi obţinut cu algoritmul de partiţionare de la `quickSort`. Dacă $j = k$, atunci problema este rezolvată. Dacă $j < k$, atunci cel de-al $k$-lea cel mai mic element trebuie căutat în subtabloul `a[j+1..n]`, iar dacă $j > k$ atunci cel de-al $k$-lea cel mai mic element trebuie căutat în subtabloul `a[1..j-1]`. Aceasta conduce la următoarea formulare recursivă a algoritmului de selectare:

```
function selecteaza(a, p, q, k)
   partitioneaza(a, p, q, k1)
   if (k1 = k)
      then return a[k]
      else if (k1 < k)
             then selecteaza(a, k1+1, q, k)
             else selecteaza(a, p, k1-1, k)
end
```

Descrierea recursivă nu este avantajoasă, deoarece produce un consum de memorie suplimentară (stiva apelurilor recursive) ce poate fi eliminat prin derecursivare:

```
function selecteaza(a, n, k)
   p ← 0
   q ← n-1
   repeat
      partitioneaza(a, p, q, k1)
      if (k1 ≠ k)
         then if (k1 < k)
                then p ← k1+1
                else q ← k1-1
   until (k1 = k)
   return a[k]
end
```

Analiza algoritmului `selecteaza` este asemănătoare cu cea a algoritmului `quickSort`. În cazul cel mai nefavorabil necesită $O(n^2)$ timp, iar pentru complexitatea medie se cunoaşte următorul rezultat:

**Teorema 10.3 ([Sed88])** *Complexitatea timp medie a algoritmului* `selecteaza` *este* $O(n)$. *Mai precis, numărul de comparaţii este aproximativ:*

$$n + k \log\left(\frac{n}{k}\right) + (n-k) \log\left(\frac{n}{n-k}\right).$$

## 10.5 Linia orizontului

### 10.5.1 Descrierea problemei

Pe o suprafaţă orizontală se consideră $n$ paralelipipede (se poate presupune că sunt clădiri), aşezate astfel încât toate paralelipipedele au o suprafaţă laterală paralelă cu o aceeaşi dreaptă. În acest fel putem considera că toate paralelipipedele au faţa spre est. O sursă de lumină, aflată la o distanţă destul de mare astfel încât razele de lumină sunt paralele şi cad perpendicular pe feţele paralelipipedelor (soarele puţin timp după ce a răsărit), proiectează paralelipipedele pe un ecran (orizontul), presupus a fi suficient de mare. Se cere să se determine conturul superior al umbrei (linia orizontului).

### 10.5.2 Modelul matematic

În primul rând să observăm că problema poate fi redusă de la o problemă în spaţiul tridimensional la o problemă în plan. Proiecţia unui paralelipiped pe ecran este un dreptunghi cu baza pe axa $Ox$. Un astfel de dreptunghi poate fi specificat prin trei elemente: cele două proiecţii $x_{stg} < x_{drp}$ ale vârfurilor pe axa $Ox$ şi înălţimea $y$. Deoarece un dreptunghi se poate afla parţial sau total în spatele altui dreptunghi, rezultă că proiecţiile se pot intersecta. În acest fel, linia orizontului va fi înfăşurătoarea superioară a unei colecţii de dreptunghiuri ce au bazele pe axa $Ox$. Pentru ca domeniul problemei să fie complet specificat, mai trebuie să vedem ce înseamnă linia orizontului. Pentru aceasta, vom răspunde mai întâi la întrebarea: din ce se compune linia orizontului? Considerăm colecţia de dreptunghiuri din figura 10.2a. Linia orizontului corespunzătoare acestei colecţii este reprezentată în figura 10.2b. Observăm că ea se compune din:

- două semidrepte incluse în axa $Ox$, care sunt extremităţile;
- segmente orizontale şi verticale care sunt laturi, porţiuni de laturi sau porţiuni din axa $Ox$;
- în plus linia orizontului are proprietatea că orice dreptunghi se află între $Ox$ şi linia orizontului.

Proiecţiile capetelor segmentelor orizontale pe axa $Ox$ determină o secvenţă de puncte, pe care le notăm de la stânga la dreapta prin $x_1, x_2, \ldots, x_k, x_{k+1}$ (s-a presupus că linia este formată de $k$ segmente orizontale). Fiecare segment orizontal $i$ are o înălţime $y_i$, $1 \leq i \leq k$. Dacă vom considera în plus $x_0 = -\infty$, $x_{k+2} = +\infty$, $y_0 = y_{k+1} = 0$, rezultă că secvenţa $x_0, y_0, x_1, y_1, \ldots, x_k, y_k, x_{k+1}, y_{k+1}, x_{k+2}$ reprezintă în mod unic linia orizontului. Pentru exemplul nostru, linia este reprezentată de secvenţa $(-\infty, x_1, y_1, x_2, 0, x_3, y_3, \ldots, x_8, 0, +\infty)$ (figura 10.3).

În general, linia orizontului poate fi reprezentată printr-o listă liniară de $2k+1$ numere reale $(x_1, y_1, \ldots, x_k, y_k, x_{k+1})$, cu $k \leq n$. Un dreptunghi poate fi considerat un caz particular de linie a orizontului ($k = 1$) şi deci poate fi reprezentat de lista liniară $(x_{stg}, y, x_{drp})$.

O colecţie de dreptunghiuri va fi reprezentată de o variabilă tablou `d`, unde `d[i]` este o structură cu câmpurile `d[i].xstg`, `d[i].xdrp`, `d[i].y`, precum şi de o variabilă de tip întreg `n`.

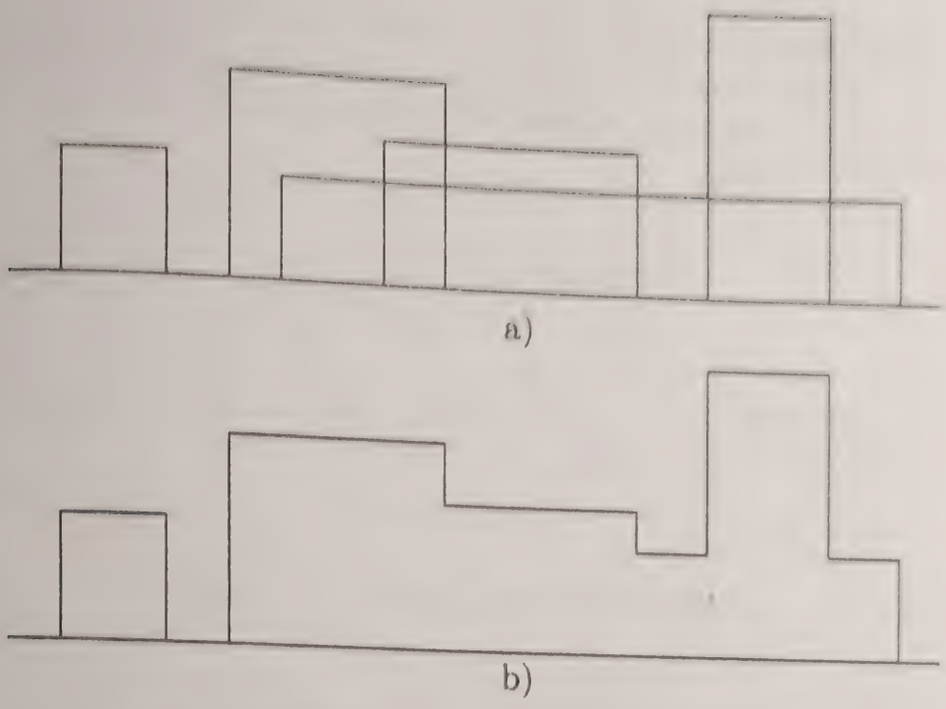



Figura 10.2: Linia orizontului

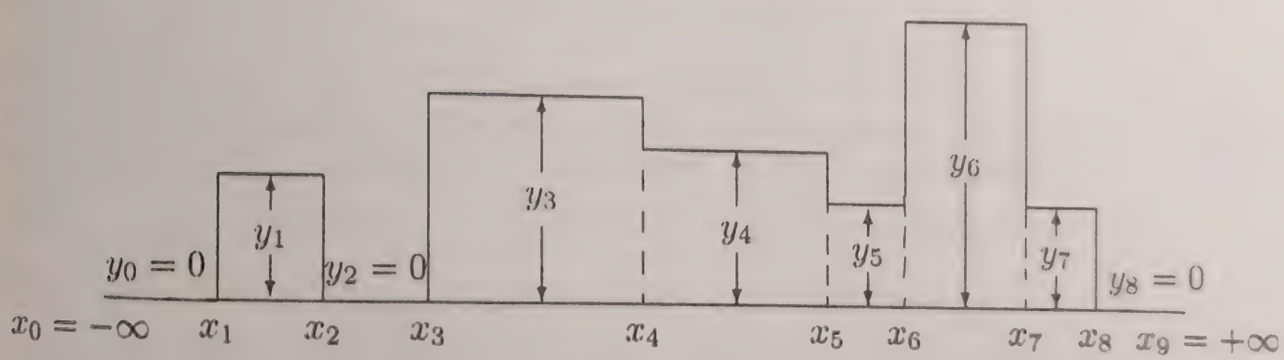



Figura 10.3: Reprezentarea liniei orizontului

Vom rezolva problema determinării liniei orizontului prin strategia *divide-et-impera*. Generalizăm presupunând date dreptunghiurile (d[p], ..., d[q]) în loc de (d[1], ..., d[n]). Există două moduri de a diviza problema:

1. *Divizarea verticală.* Se consideră un $z$ cu $d[p].xstg \leq z \leq d[p].xdrp$. O subproblemă va fi formată din colecţia de dreptunghiuri aflate la stânga lui $z$ şi cealaltă subproblemă din colecţia de dreptunghiuri aflate la dreapta lui $z$. Dreptunghiurile $d[i]$ cu $d[i].stg < z < d[i].drp$ (o latură verticală la stânga lui $z$ şi cealaltă la dreapta lui $z$) vor fi divizate în două dreptunghiuri prin considerarea unei laturi verticale duse prin $z$. Această strategie are avantajul că partea de asamblare este simplă (constă din concatenarea celor două linii ale orizontului obţinute prin rezolvarea subproblemelor), dar are şi marele dezavantaj ca sunt cazuri când pasul de divizare nu conduce la micşorarea numărului de dreptunghiuri pentru subprobleme. Ca exemplu, considerăm următoarea colecţia de două dreptunghiuri din figura 10.4a. Indiferent cum ar fi ales $z$ între $x_1$ şi $x_4$ (figura 10.4b), cel puţin una din subprobleme are

două sau mai multe dreptunghiuri (atenţie, procesul de divizare poate conduce chiar la mărirea numărului de dreptunghiuri într-o subproblemă).
De remarcat totuşi că o ordonare lexicografică conduce la o simplificare a pasului de divizare.

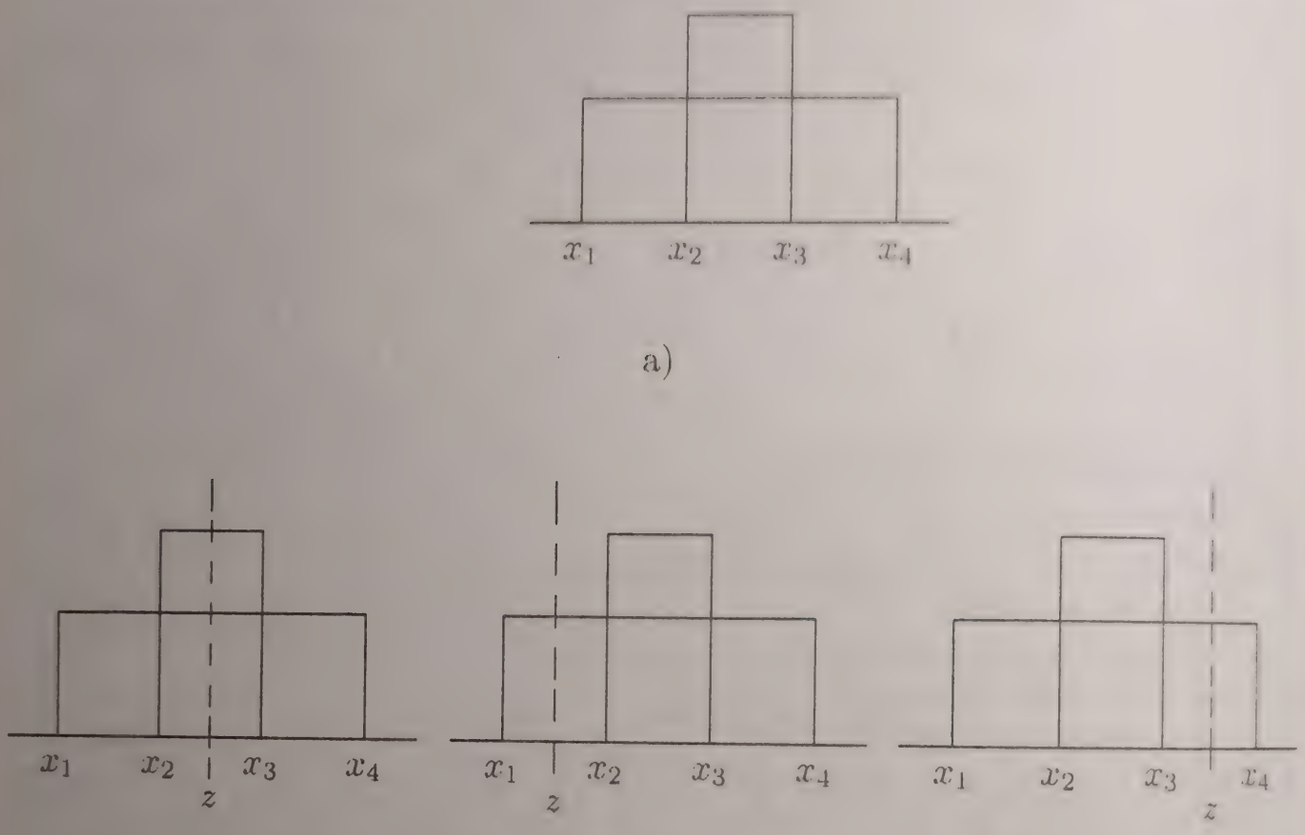



Figura 10.4: Un exemplu când divizarea verticală nu reduce numărul de dreptunghiuri

2. *Divizarea orizontală.* Pasul de divizare este foarte simplu: prima subproblemă este formată din primele $\left[\frac{n}{2}\right]$ dreptunghiuri, iar a două subproblemă din celelalte $n - \left[\frac{n}{2}\right]$ dreptunghiuri. Nu mai este necesară ordonarea dreptunghiurilor. Acum este suficient să se determine $m = \left[\frac{p+q}{2}\right]$ şi prima subproblemă va conţine colecţia (sublista) $d[p], \ldots, d[m]$, iar a doua subproblemă va conţine colecţia (sublista) $d[m+1], \ldots, d[q]$. Partea de asamblare este un pic mai dificilă pentru că necesită interclasarea celor două linii obţinute prin rezolvarea subproblemelor.

Divizarea orizontală pare mai potrivită, motiv pentru care o alegem pentru determinarea liniei orizontului. Mai întâi considerăm un exemplu. Fie colecţia din figura 10.5. Prin divizare se obţin subproblemele reprezentate în figura 10.6. Rezolvarea subproblemelor conduce la soluţiile din figura 10.7 care interclasate dau linia reprezentată în figura 10.8.

În continuare vom descrie modul în care se realizează interclasarea. Presupunem că prin rezolvarea subproblemelor s-au obţinut liniile `L1 : TLinieOrizont` cu valoarea

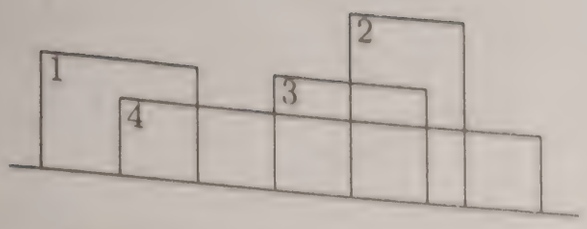



Figura 10.5: Problema inițială

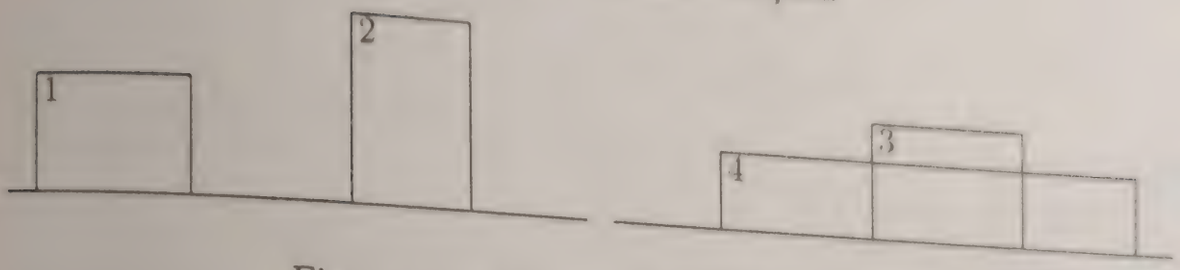



Figura 10.6: Divizarea în subprobleme

$(x_1, y_1, \ldots, x_{m1}, y_{m1}, x_{m1+1})$, respectiv `L2:TLinieOrizont` cu valoarea $(x'_1, y'_1, \ldots, x'_{m2}, y'_{m2}, x'_{m2+1})$. Procesul de interclasare are loc comparând două câte două componentele orizontale ale celor două linii şi alegându-o de fiecare dată pe cea cu înălţimea mai mare. Notăm că termenul „*interclasare*" este utilizat corect: cele două linii `L1` şi $L2$ sunt liste ordonate de segmente orizontale, iar rezultatul este tot o listă ordonată de segmente orizontale. Lungimea curentă a liniei rezultat este memorată de variabila `m`. Presupunem că la momentul curent se compară $L1[i] = x_{[\frac{i}{2}]+1}$ şi $L2[j] = x'_{[\frac{j}{2}]+1}$. Pentru a simplifica scrierea, notăm $\left[\frac{i}{2}\right] + 1$ cu $i_1$ şi $\left[\frac{j}{2}\right] + 1$ cu $j_1$. De asemenea, notăm cu $h_1$ înălţimea curentă din `L1`, cu $h_2$ înălţimea curentă din `L2`, cu $h$ înălţimea curentă din linia rezultat `L` şi cu `h1`, `h2`, respectiv `h` variabilele care memorează aceste înălţimi. O variabilă `k` va indica acea linie care contribuie în pasul curent la linia rezultat: $k = 1$ indică faptul că linia `L1` contribuie la formarea liniei `L` şi $k = 2$ indică faptul că linia $L2$ contribuie la formarea liniei `L`. Distingem următoarele situaţii:

1. $x_{i_1} < x'_{j_1}$. În continuare, $h_1$ va deveni egală cu $L1[i+1]$. Dacă $i \geq 2 * m1 + 1$, atunci $h_1$ va deveni zero. Valoarea variabilei `j` rămâne neschimbată. Se disting subcazurile:
   a) $h_1 < h_2$. Dacă $k = 1$ (figura 10.9a), atunci $h > h_2$ şi în continuare vom avea:
   `k` ← 2

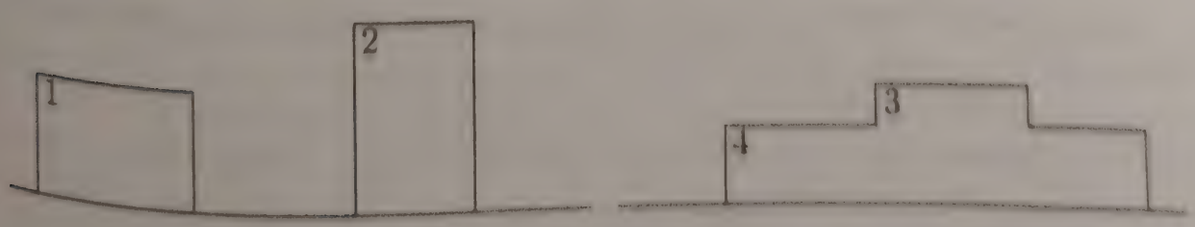



Figura 10.7: Rezolvarea subproblemelor

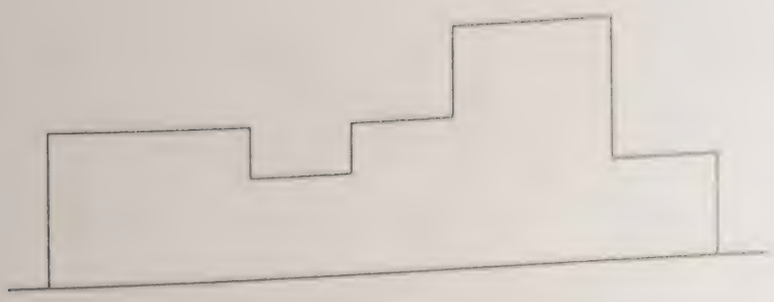
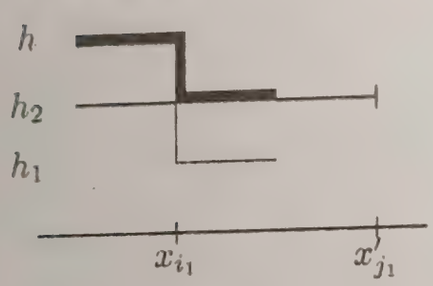
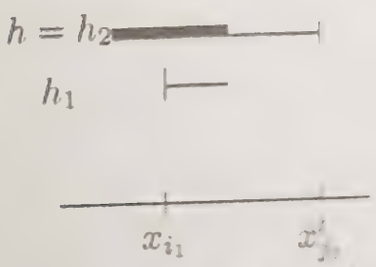


Figura 10.8: Asamblarea soluţiilor

```
L[m+1] ← L1[i]
L[m+2] ← h2
h ← h2
m ← m+2
i ← i+2
```




a) $x_{i_1} < x'_{j_1}$ şi $h_1 < h_2$




b) $x_{i_1} < x'_{j_1}$ şi $h = h_2$

Figura 10.9: Cazul 1

Dacă $k = 2$ (figura 10.9b), atunci $h = h_2$ şi în continuare k, m şi h rămân neschimbate. Singura modificare care este i ← $i + 2$.

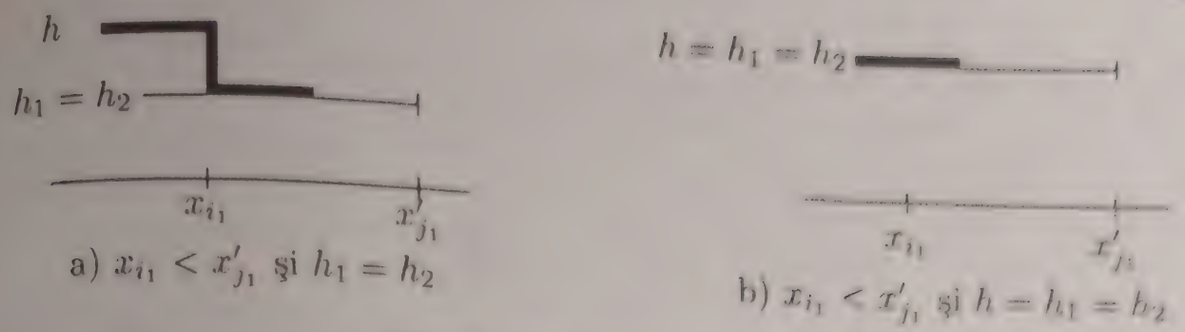
b) $h_1 = h_2$ . Dacă $k = 1$ (figura 10.10a), atunci $h > h_2$ şi în continuare vom avea:

```
L[m+1] ← L1[i]
L[m+2] ← h1
h ← h1
m ← m+2
i ← i+2
```

iar k rămâne neschimbată. Dacă $k = 2$ (figura 10.10b), atunci $h = h_2$ şi în continuare k, m, h rămân neschimbate. Singura modificare este i ← $i + 2$.

c) $h_1 > h_2$. Dacă $k = 1$ (figura 10.11a), atunci $h > h_2$ şi în continuare vom avea:

```
L[m+1] ← L1[i]
L[m+2] ← h1
h ← h1
m ← m+2
i ← i+2
```

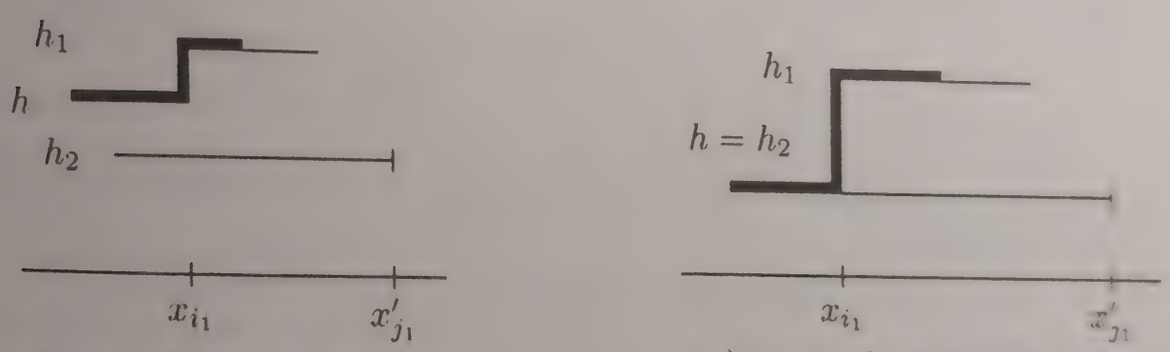



a) $x_{i_1} < x'_{j_1}$ şi $h_1 = h_2$ b) $x_{i_1} < x'_{j_1}$ şi $h = h_1 = h_2$

Figura 10.10: Cazul 1

iar k rămâne neschimbată. Dacă $k = 2$, atunci $h = h_2$ (figura 10.11b) şi vom avea:

```
k ← 1
L[m+1] ← L1[i]
L[m+2] ← h1
h ← h1
m ← m+2
i ← i+2
```




a) $x_{i_1} < x'_{j_1}$ şi $h, h_1 > h_2$ a) $x_{i_1} < x'_{j_1}$ şi $h_1 > h = h_2$

Figura 10.11: Cazul 1

Cazurile $x_{i_1} = x'_{j_1}$ şi $x_{i_1} > x'_{j_1}$ se analizează la fel. Considerăm drept cazuri de bază următoarele două:

- Colecţia nu are dreptunghiuri ($n = 0$). În acest caz, linia orizontală coincide cu axa $Ox$.
- Colecţia are un singur dreptunghi ($n = 1$). Linia orizontului coincide cu reprezentarea dreptunghiului.

### 10.5.3 Implementare

Programul complet care determină linia orizontului se obţine prin asamblarea secvenţelor de mai sus. De notat că anumite variabile pot fi eliminate. De exemplu, se poate ţine cont de faptul că $h1 = L1[i-1], h2 = L2[j-1], h = L[m]$. Considerând elementele suplimentare $L1[0] = L2[0] = L[0] = 0$, se poate renunţa la utilizarea variabilelor h1, h2 şi h.

Aplicând teorema 10.1 pentru cazul $a = b = 2$ şi $k = 1$ obţinem că algoritmul de determinare a liniei orizontului are complexitatea timp $O(n \log n)$.

**Exerciţiul 10.5.1.** Descrierea algoritmului cu ajutorul unui subprogram recursiv are dezavantajul că utilizează două tablouri locale pentru memorarea liniilor rezultate din rezolvarea subproblemelor. Salvarea acestor tablouri în stiva sistemului reduce semnificativ performanţele programului. Să se elimine acest inconvenient printr-un procedeu asemănător cu cel de la sortare prin interclasare.

## 10.6 Transformata Fourier discretă

### 10.6.1 Modelul matematic

Un semnal poate fi văzut din două puncte de vedere diferite:

1. domeniul timp;
2. domeniul frecvenţă.

Notăm cu $f(t)$ funcţia care descrie semnalul în domeniul timp şi cu $F(\nu)$ funcţia care descrie semnalul în domeniul frecvenţei. Se poate trece dintr-o descriere în alta prin *transformata Fourier*. Trecerea din domeniul timp în domeniul frecvenţă se face prin *transformata Fourier directă*:

$$F(\nu) = \mathcal{F}[f(t)] = \int_{-\infty}^{+\infty} f(t)\mathrm{e}^{-2\pi i\nu t}dt, \tag{10.5}$$

iar trecerea de la domeniul frecvenţă la domeniul timp se face prin *transformata Fourier inversă*:

$$f(t) = \mathcal{F}^{-1}[F(\nu)] = \int_{-\infty}^{+\infty} F(\nu)\mathrm{e}^{2\pi i\nu t}d\nu. \tag{10.6}$$

*Transformata Fourier discretă* se obţine când se consideră $f(t)$ măsurată într-un număr finit de puncte: $t_0, \ldots, t_{n-1}$ cu $t_j = jT$, unde $T$ este perioada de timp la care se fac măsurătorile. Notând $x_k = f(t_k)$, pentru $k = 0, \ldots, n-1$, *transformata Fourier discretă directă* este dată de relaţia:

$$\mathcal{F}[f(t)] = \sum_{k=0}^{n-1} x_k \mathrm{e}^{-2\pi i \frac{j}{nT}kT}T = T\sum_{k=0}^{n-1} x_k \mathrm{e}^{\frac{-2\pi ijk}{n}}. \tag{10.7}$$

Notăm:

$$W_j = \sum_{k=0}^{n-1} x_k \mathrm{e}^{\frac{-2\pi ijk}{n}}. \tag{10.8}$$

*Transformata Fourier discretă inversă* este dată de relaţia:

$$x_k = \sum_{j=0}^{n-1} W_j T \mathrm{e}^{\frac{2\pi ijk}{n}}\frac{1}{nT} = \frac{1}{n}\sum_{j=0}^{n-1} W_j \mathrm{e}^{\frac{2\pi ijk}{n}} \tag{10.9}$$

Dacă se consideră polinomul de variabilă $Y$:

$$f(Y) = x_0 + x_1 Y + \cdots x_{n-1}Y^{n-1} \tag{10.10}$$

şi rădăcinile de ordinul $n$ ale unităţii:

$$w_j = e^{\frac{2\pi ij}{n}}, j = 0, 1, \ldots, n-1,$$

obţinem

$$W_{n-j} = f(w_j), j = 0, 1, \ldots, n-1. \quad (10.11)$$

unde am considerat $W_n = W_0$.

Rădăcinile de ordinul $n$ ale unităţii satisfac următoarele proprietăţi:

**Lema 10.1.** *Dacă $n$ este par, $n = 2m$, şi $j < m$, atunci $-w_j = w_{j+m}$.*

*Demonstraţie.* Avem $w_{j+m}^2 = (e^{\frac{2\pi i(j+m)}{2m}})^2 = (e^{\frac{2\pi ij}{2m}})^2 (e^{\frac{2\pi im}{2m}})^2 = (e^{\frac{2\pi ij}{2m}})^2 e^{\frac{2\pi i 2m}{2m}} = w_j^2$. Deoarece $w_j$ şi $w_{j+m}$ sunt distincte, rezultă $w_j = -w_{j+m}$. sfdem

**Lema 10.2.** *Dacă $n$ este par, $n = 2m$, şi $1 \leq j < m$, atunci $w_j^2$ este de asemenea o rădăcină $\neq 1$.*

*Demonstraţie.* Deoarece $w_j^n = 1$ rezultă $(w_j^2)^n = 1$. Pe de altă parte, $w_j^2 = (e^{\frac{2\pi ij}{n}})^2 = e^{\frac{2\pi i(j+j)}{n}} = w_{j+j} \neq 1$ pentru că $2 \leq j + j \leq n - 2$. sfdem

În continuare ne ocupăm de un mijloc de calcul eficient al valorilor $f(w_j)$. Reamintim că determinarea valorii unui polinom într-un punct se poate realiza prin $n$ înmulţiri şi $n$ adunări, utilizând schema lui Horner (a se vedea exerciţiul 10.2), şi aceste valori sunt optime [HS84, Kro79]. De aici, calculul direct al celor $n$ componente ale transformatei Fourier necesită timpul $O(n^2)$. Utilizând strategia *divide-et-impera*, vom proiecta un algoritm care necesită $O(n \log n)$ timp. Numim calculul dat de acest algoritm *transformata Fourier rapidă* şi-l notăm FFT (Fast Fourier Transform). Considerăm mai întâi cazul $n = 2^r$. Utilizăm proprietăţile de mai sus ale rădăcinilor unităţii pentru a despărţi suma din definiţia lui $f(w_j)$ în două subsume: suma coeficienţilor pari şi suma coeficienţilor impari. Avem:

$$\begin{aligned} f(w_j) &= \sum_{m=0}^{\frac{n}{2}-1} x_{2m} w_j^{2m} + \sum_{m=0}^{\frac{n}{2}-1} x_{2m+1} w_j^{2m+1} \\ &= \sum_{m=0}^{\frac{n}{2}-1} x_{2m} w_j^{2m} + w_j \sum_{m=0}^{\frac{n}{2}-1} x_{2m+1} w_j^{2m} \end{aligned}$$

De fapt, expresia de mai sus corespunde scrierii polinomului $f(t)$ sub forma:

$$f(Y) = (x_0 + x_2 Y^2 + \cdots + x_{n-2} Y^{n-2}) + Y(x_1 + x_3 Y^2 + \cdots + x^{n-1} Y^{n-2}) = g(Y) + Y h(Y)$$

unde $g(Y)$ şi $h(Y)$ sunt polinoame de grad $n - 2$, *i.e.*, de grad mai mic cu 1 decât cel al lui $f(Y)$. Evaluarea acestor polinoame se va face numai în punctele $w_m, m = 0, 1, \ldots, \frac{n}{2} - 1 = 2^{r-1} - 1$. Considerăm listele de numere $x_0, x_2, x_4, \ldots, x_{2^r-2}$ respectiv $x_1, x_3, x_5, \ldots, x_{2^r-1}$. Seriile Fourier ale acestor liste constau din valorile polinoamelor:

$$g'(Y) = x_0 + x_2 Y + x_4 Y^2 + \cdots + x_{n-2} Y^{\frac{n}{2}-1}$$

şi

$$h'(Y) = x_1 + x_3 Y + x_5 Y^2 + \cdots + x_{n-1} Y^{\frac{n}{2}-1}$$

pentru rădăcinile de ordinul $\frac{n}{2}$ ale unităţii. Notăm cu $w'_j$, $j = 0, 1, \ldots, \frac{n}{2} - 1$ aceste rădăcini. Se pune problema dacă putem calcula $f(w_j)$, utilizând $g'(w'_j)$ şi $h'(w'_j)$. Pentru $j < \frac{n}{2}$ avem:

$$w_j^{2m} = e^{\frac{2\pi i j 2m}{2^r}} = e^{\frac{2\pi i j m}{2^{r-1}}} = {w'_j}^m$$

iar pentru $j \geq \frac{n}{2}$ are loc:

$$w_j^{2m} = -w_{j-\frac{n}{2}}^{2m} = -e^{\frac{2\pi i (j-\frac{n}{2}) 2m}{2^r}} = e^{\frac{2\pi i (j-\frac{n}{2}) m}{2^{r-1}}} = {w'}_{j-\frac{n}{2}}^m$$

Aceste relaţii arată că valorile $f(w_j)$ pot fi calculate dacă se cunosc $g'(w'_j)$ şi $h'(w'_j)$: $f(w_j) = g'(w'_j) + w_j h'(w'_j)$, dacă $j < \frac{n}{2}$ şi $f(w_j) = g'(-w'_j) + w_j h'(w'_{j-n})$ dacă $j \geq \frac{n}{2}$. Deci strategia *divide-et-impera* poate fi aplicată cu succes. Algoritmul rezultat este:

```
procedure FFT(f, n, w)
   if (n = 1)
      then f(w0) ← x0
      else n ← ⌈n/2⌉
           g' ← x0 + x2Y + ··· + x2n-2Y^n-1
           h' ← x1 + x3Y + ··· + x2n-1Y^n-1
           FFT(g', n, w)
           FFT(h', n, w)
           for j ← 0 to n-1 do
               f(wj) ← g'(w'j) + wj * h'(w'j)
           for j ← n to 2*n-1 do
               f(wj) ← g'(w'j-n) + wj * h'(w'j-n)
end
```

În algoritmul de mai sus există dezavantajul că trebuie calculate atât rădăcinile unităţii de ordinul $n$, cât şi cele de ordinul $\frac{n}{2}$. Acest dezavantaj poate fi eliminat dacă se schimbă instanţa problemei. Plecând de la observaţia că dacă se cunoaşte o rădăcină primitivă de ordinul $n$ a lui 1, să zicem $w \neq 1$, atunci celelalte rădăcini sunt puteri ale lui $w$. Deci putem lua $w_j = w^j$. Instanţa problemei se modifică în felul următor: în loc să considerăm dat numai polinomul $f$, presupunem date polinomul $f$ de grad $n-1$ şi o rădăcină primitivă de ordinul $n$ a unităţii. Ieşirea constă în determinarea valorilor $f(w^0), f(w^1), f(w^2), \ldots, f(w^{n-1})$. De asemenea, se ţine cont de faptul că dacă $w$ este o rădăcină primitivă de ordinul $n$ a unităţii, atunci $w^2$ este o rădăcină primitivă de ordinul $\frac{n}{2}$ a unităţii. Noul algoritm este:

```
procedure FFT(f, n, w)
   if (n = 1)
      then f(w0) ← x0
      else n ← ⌈n/2⌉
```

```
        g' ← x0 + x2Y + ··· + x2n-2Y^n-1
        h' ← x1 + x3Y + ··· + x2n-1Y^n-1
        FFT(g', n, w^2)
        FFT(h', n, w^2)
        for j ← 0 to n-1 do
            f(w^j) ← g'(w^2j) + w * h'(w^2j)
        for j ← n to 2*n-1 do
            f(w^j) ← g'(w^2j) + w * h'(w^2j)
    end
```

**Teorema 10.4.** *Dacă $n$ este o putere a lui 2, atunci transformata Fourier discretă pentru $n$ măsurători poate fi calculată în timpul $O(n \log n)$.*

*Demonstrație.* Se aplică teorema 10.1 pentru $a = 2 = b$ și $k = 1$. ∎

## 10.6.2 Implementare

Următoarele funcții descriu operații peste numere complexe:

- adunarea:

```
function ad(x,y)
    z.re ← x.re + y.re
    z.im ← x.im + y.im
    return z
end
```

- scăderea:

```
function sc(x,y)
    z.re ← x.re - y.re
    z.im ← x.im - y.im
    return z
end
```

- înmulțirea:

```
function inm(x,y)
    z.re ← x.re*y.re - x.im*y.im
    z.im ← x.re*y.im + x.im*y.re
    return z
end
```

Procedura recursivă care implementează schema FFT este următoarea:

```
procedure FFT(f, n, w, fw)
    if (n = 1)
        then fw[0] ← f[0]
        else n ← n / 2
             for i ← 0 to n-1 do
                 g[i] ← f[2*i]
                 h[i] ← f[2*i+1]
             inm(w,w,w1)
```

```
        FFT(g,n,w1,gw)
        FFT(h,n,w1,hw)
        w1 ← complex(1)
        for j ← 0 to n-1 do
            w2 ← inm(w1,hw[j])
            fw[j] ← ad(gw[j],w2)
            fw[j+n] ← sc(gw[j],w2)
            w1 ← inm(w1,w)
    end
```

## 10.7 Exerciţii

**Exerciţiul 10.1.** Se ştie că maximul (minimul) dintr-o listă cu $n$ elemente se poate determina făcând $n-1$ comparaţii. Astfel, se poate scrie un program care calculează simultan şi maximul şi minimul făcând $2(n-1)$ comparaţii. Să se proiecteze un algoritm *divide-et-impera* care determină simultan minimul şi maximul. Algoritmul va trebui să execute $\frac{3n}{2} - 2$ comparaţii. dacă $n$ este o putere a lui 2.

**Exerciţiul 10.2.** (*Evaluarea polinoamelor*) Să se scrie un algoritm *divide-et-impera* care să calculeze valoarea unui polinom într-un punct. Care este eficienţa algoritmului? Să se compare aceasta cu cea a algoritmului bazat pe schema lui Horner:

$$f(x) = a_0 + (a_1 + \cdots (a_n x + a_{n-1})x + \cdots)x$$

**Exerciţiul 10.3.** [MS91] (*Numere Fibonacci*) Să se scrie un program care calculează al $n$-lea număr Fibonacci $F(n)$ executând $\Theta(\log n)$ operaţii aritmetice.

**Exerciţiul 10.4.** [MS91] (*Înmulţire rapidă*)

1. Arătaţi că două polinoame de gradul 1 pot fi înmulţite executând exact trei înmulţiri.
2. Utilizând 1, să se scrie un algoritm *divide-et-impera* care să înmulţească două polinoame de grad $n$ în timpul $\Theta(n^{\log_2 3})$.
3. Utilizând 2, să se arate că doi întregi reprezentaţi binar pe $n$ biţi pot fi înmulţiţi în timpul $\Theta(n^{\log_2 3})$, unde orice operaţie binară invocă un număr constant de biţi.

**Exerciţiul 10.5.** *Diametrul* unui arbore (graf conex fără cicluri) este lungimea celui mai lung drum între două vârfuri. Să se găsească un algoritm *divide-et-impera* pentru determinarea diametrului unui arbore.

**Exerciţiul 10.6.** (*Algoritmul lui Strassen de înmulţire a matricilor*) Înmulţirea a două matrici $A$ şi $B$ se poate realiza cu numai şapte înmulţiri:

- se calculează expresiile:

$$\begin{aligned}
q_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
q_2 &= (a_{21} + a_{22})b_{11} \\
q_3 &= a_{11}(b_{12} - b_{22}) \\
q_4 &= a_{22}(-b_{11} + b_{21}) \\
q_5 &= (a_{11} + a_{12})b_{22} \\
q_6 &= (-a_{11} + a_{21})(b_{11} + b_{12}) \\
q_7 &= (a_{12} + -a_{22})(b_{21} + b_{22})
\end{aligned}$$

- apoi se calculează elementele matricei $C = AB$:

$$c_{11} = q_1 + q_4 - q_5 + q_7$$
$$c_{12} = q_3 + q_5$$
$$c_{21} = q_2 + q_4$$
$$c_{22} = q_1 + q_3 - q_2 + q_6$$

Să se arate că înmulţirea a două matrici $n \times n$ se poate face cu $O(n^{\log_2 7})$ înmulţiri.

**Exerciţiul 10.7.** Se consideră un ecran alb/negru cu rezoluţia $1024 \times 1024$. Noţiunea de fereastră este definită recursiv astfel:

- ecranul este o fereastră;
- un pixel este o fereastră;
- o fereastră de dimensiune $n \times n$ defineşte 4 ferestre (stânga-sus, dreapta-sus, stânga-jos şi dreapta-jos) de dimensiuni $\frac{n}{2} \times \frac{n}{2}$.

Colorarea ferestrelor se poate realiza cu următorul set de instrucţiuni:

`@` - şterge ecranul,
`a` - subfereastra stânga-sus devine fereastră curentă,
`b` - subfereastra dreapta-sus devine fereastră curentă,
`c` - subfereastra dreapta-jos devine fereastră curentă,
`d` - subfereastra stânga-jos devine fereastră curentă,
`f` - ultima fereastră vizitată înaintea celei curente devine fereastră curentă.

Se cere:

1. Să se arate că pentru orice dreptunghi de pe ecran, cu laturile paralele cu marginile ecranului, există o secvenţă de instrucţiuni care realizează colorarea dreptunghiului.
2. Să se proiecteze un algoritm care, pentru un astfel de dreptunghi dat, determină o secvenţă de instrucţiuni de lungime minimă care realizează colorarea dreptunghiului.

## 10.8 Referinţe bibliografice

Prezentarea de aici se bazează pe [HS84, MS91, LG86]. În [Sed88], metoda este prezentată ca o aplicaţie a recursiei.

# Capitolul 11

# Programare dinamică

## 11.1 Exemplu: drum optim într-o reţea piramidală de numere

Considerăm următoarea problemă foarte simplă. Fie $a = (a_0, \ldots, a_{n-1})$ o secvenţă de numere întregi, astfel încât $n + 1 = \frac{h(h+1)}{2}$, şi $R(a)$ reţeaua piramidală ce are nodurile etichetate cu numerele $a_i$ aşa cum este sugerat în figura 11.1. Se pune problema determinării unui drum din vârful piramidei la baza piramidei pentru care suma numerelor aflate pe drum este maximă. Dacă notăm cu $D$ mulţimea drumurilor de la vârf la bază şi cu $f(d)$ suma numerelor aflate pe drumul $d$ atunci funcţia obiectiv este:

$$\max_{d \in D} f(d)$$

Prin stare a problemei vom înţelege subproblema $DMR(i)$ corespunzătoare subreţelei cu vârful în $a_i$. Funcţia obiectiv a subproblemei $DMR(i)$ este:

$$\max_{d \in D(i)} f(d)$$

unde $D(i)$ este mulţimea drumurilor care pleacă din vârful $a_i$ spre baza piramidei. Notăm cu $f(i)$ valoarea funcţiei obiectiv pentru $DMR(i)$. Fie $st(i)$ indicele fiului stânga al lui $a_i$ şi $dr(i)$ indicele fiului dreapta al lui $a_i$. Există relaţia evidentă $dr(i) = st(i) + 1$. Drumul optim ce pleacă din $a_i$ va trece prin unul din cei doi fii ai lui $a_i$. Presupunem că trece prin $a_{st(i)}$. Subdrumul ce pleacă din $st(i)$ este la rândul său optim. Se raţionează asemănător în cazul când drumul optim trece prin $a_{dr(i)}$ şi rezultă următoarea relaţie de recurenţă:

$$f(i) = \max(f(st(i)), f(dr(i))) + a_i \tag{11.1}$$

Relaţia 11.1 poate fi dovedită utilizând inducţia şi reducerea la absurd. Dacă se rezolvă recursiv această relaţie, atunci $f(4)$ va fi calculată de două ori, $f(7)$ şi $f(8)$ de trei ori ş.a.m.d. De aceea valorile corespunzătoare drumurilor optime

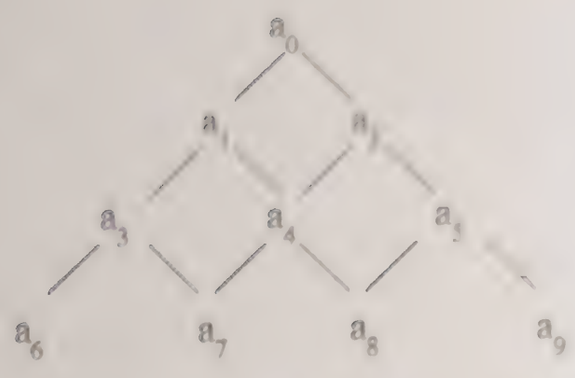



Figura 11.1: $R(a)$ pentru $n = 10$

vor fi memorate într-un tablou unidimensional şi ordinea de calcul va fi de la baza piramidei spre vârf, adică în ordinea descrescătoare a indicilor.

Drumul optim $b_0, b_1, \ldots$ pentru o reţea de numere este determinat, pe baza tabloului cu valori optime, prin următorul algoritm:

```
b[0] ← a[0];
for j ← 1 to k do
  if (f[i]=f[st(i)]+b[j-1])
  then b[j] ← st(i)
  else b[j] ← dr(i)
```

## 11.2 Prezentarea intuitivă a paradigmei

Principalele ingrediente ale paradigmei programare dinamică sunt următoarele:

1. *Clasa de probleme* la care se aplică include probleme de optim.
2. Definirea noţiunii de *stare*, care este de fapt o subproblemă, şi asocierea funcţiei obiectiv pentru stare (subproblemă).
3. Definirea unei relaţii de tranziţie între stări. O relaţie $s \rightarrow s'$, unde $s$ şi $s'$ sunt stări, va fi numită *decizie*. O *politică* este o secvenţă de decizii consecutive, adică o secvenţă de forma $s_0 \rightarrow s_1 \rightarrow \cdots s_n$.
4. Aplicarea *Principiului de optim* pentru obţine o relaţie de recurenţă. Principiul de optim (PO) afirmă că o subpolitică a unei politici optimale este la rândul ei optimală. Deoarece este posibil ca PO să nu aibă loc, rezultă că trebuie verificată validitatea relaţiei de recurenţă.
5. *Calculul recurenţei* rezolvând subproblemele de la mic la mare şi memorând valorile obţinute într-un tablou. Nu se recomandă scrierea unui program recursiv care să calculeze valorile optime. Dacă în procesul de descompunere problemă $\hookrightarrow$ subproblemă, o anumită subproblemă apare de mai multe ori, ea va fi calculată de algoritmul recursiv de câte ori apare.
6. Extragerea soluţiei optime din tablou utilizând proprietatea de *substructură optimă a soluţiei*, care afirmă că *soluţia optimă a problemei include soluţiile optime ale subproblemelor sale*. De remarcat că proprietatea de substructură optimă este echivalentă cu principiul de optim.

Ca şi în cazul celorlalte paradigme, aplicarea programării dinamice se poate face în două moduri:

- direct: se defineşte noţiunea de *stare* (de cele mai multe ori ca fiind o subproblemă), se aplică principiul de optim pentru a deduce relaţiile de recurenţă de tip 11.8 şi apoi se stabilesc strategiile de calcul al valorilor şi soluţiilor optime;
- prin comparare: se observă că problema este asemănătoare cu una dintre problemele cunoscute şi se încearcă aplicarea strategiei în aceeaşi manieră ca în cazul problemei corespunzătoare.

## 11.3 Alocarea resurselor

### 11.3.1 Descrierea problemei

Pentru realizarea a $p$ proiecte sunt disponibile $r$ resurse. Alocarea a $j$ resurse la proiectul $i$ produce un profit $c[i, j]$. Problema constă în alocarea celor $r$ resurse, astfel încât profitul total să fie maxim.

Problema poate fi reformulată ca o problemă de optim peste digrafuri. Considerăm mai întâi un exemplu.

**Exemplu.** Presupunem $p = 3$ şi $r = 2$. Digraful asociat problemei este reprezentat în figura 11.2. Există un vârf iniţial $s$, care corespunde numărului iniţial de resurse, şi un vârf final $t$, care corespunde numărului de resurse rămase nealocate (care este zero). Un vârf intermediar corespunde numărului de resurse ce pot fi alocate în continuare. Arcele sunt etichetate cu profiturile aduse de resursele ce pot fi alocate unui anumit proiect. Nu pot fi alocate mai multe resurse decât sunt disponibile. Soluţia optimă descrie un drum de la $s$ la $t$. sfex

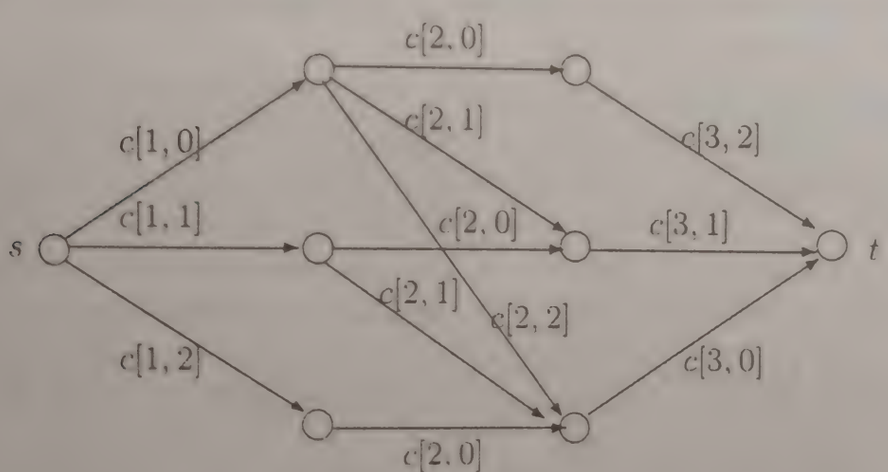



Figura 11.2: Alocarea resurselor -- reprezentarea cu digrafuri

### 11.3.2 Modelul matematic

Construcţia digrafului asociat problemei pentru cazul general poate fi dată în modul următor: mulţimea de vârfuri ale digrafului este partajată în $p + 1$ submulţimi disjuncte: $V_1, \ldots, V_{p+1}$. $V_1$ conţine un singur vârf $s$, $V_{p+1}$ conţine tot un singur

vârf $t$ şi $V_i$ conţine $r+1$ vârfuri, $(i,j)$, $0 \le j \le r-1 < i \le p$. Pentru uniformizare, vom presupune $s = (1,r)$, $t = (p+1,0)$. Un arc uneşte un vârf $(i,j) \in V_i$ cu un vârf $(i+1,k) \in V_{i+1}$ numai dacă $j \ge k$ şi în acest caz este etichetat cu $c[i,j-k]$. Semnificaţia unui arc $((i,j),(i+1,k))$ cu eticheta $c[i,j-k]$ este următoarea: dacă la pasul $i$ sunt disponibile $j$ resurse şi la proiectul $i$ se alocă $j-k$ resurse, atunci se obţine un profit $c[i,j-k]$ şi rămân $k$ resurse disponibile. Un arbel de graf este un caz particular de digraf etajat (*multi-stage graph*) definit după cum urmează:

**Definiţia 11.1.** *Un* digraf etajat *este un digraf $G = (V,E)$, în care $V = V_1 \cup \cdots \cup V_{p+1}$, cu proprietăţile:*

- *$V_i \cap V_j = \emptyset$ dacă $i \ne j$.*
- *$\#V_1 = \#V_{p+1} = 1$. Presupunem $V_1 = \{s\}, V_{p+1} = \{t\}$.*
- *dacă un arc are sursa în $V_i$, atunci are destinaţia în $V_{i+1}$.*

Problema alocării resurselor poate fi generalizată acum la o problemă de optim peste digrafuri etajate.

> Fie date un digraf etajat $G = (V,A)$ şi o funcţie de profit (cost) $c : A \to \mathbb{R}$. Profitul (costul) unui drum este suma profiturilor (costurilor) arcelor care descriu drumul. Problema constă în determinarea unui drum de la $s$ la $t$ cu profit maxim (cost minim).

Presupunem $V = \{0,1,\ldots,n-1\}$, $s = 0$, $t = n-1$ şi că vârfurile de pe acelaşi etaj sunt numere consecutive.

Vom încerca să explicăm parcurgerea fiecărei etape a programării dinamice. Definim drept stare problema determinării drumurilor optime de la orice vârf $j \in X \subseteq V$ la $t$. Notăm această problemă prin DODE($X$) (Drumurile Optime în Digrafuri Etajate). Pentru $i$ $(1 \le i \le p+1)$ şi $j \in V_i$ notăm cu $D[i,j]$ drumul optim de la vârful $j$ la $t$ şi cu ValOpt$[i,j]$ profitul (costul) acestui drum. ValOpt$[i,j]$ reprezintă valoarea asociată stării DODE($\{j\}$). Valoarea asociată stării DODE($X$) va fi mulţimea $\{\text{ValOpt}[i,j] \mid j \in X\}$.

Ne punem problema găsirii relaţiei analitice pentru valoarea optimă ValOpt$[i,j]$. Dacă $i = p+1$, atunci singura valoare posibilă pentru $j$ este $n$ şi avem ValOpt$[p+1,n] = 0$. Considerăm $i < p$. Orice drum de la $j$ la $n$ începe cu un arc cu destinaţia în $V_{i+1}$. Notăm cu $k$ destinaţia primului arc de pe drumul optim $D[i,j]$. Selectarea acestui arc poate fi gândită ca fiind decizia prin care starea DODE($\{j\}$) este transformată în starea DODE($\{k\}$). Aplicând principiul de optim, politica optimă aplicată stării DODE($\{k\}$) este o subpolitică a politicii optime corespunzătoare stării DODE($\{j\}$). Rezultă ValOpt$[i,j] = c[j,k] + \text{ValOpt}[i+1,k]$. De aici, rezultă următoarea ecuaţie pentru ValOpt:

$$\begin{aligned} &\text{ValOpt}[p+1,n] = 0 \\ &\text{ValOpt}[i,j] = \text{optim}\{c[j,k] + \text{ValOpt}[i+1,k] \mid k \in V_{i+1}, \langle j,k\rangle \in A\} \end{aligned} \tag{11.2}$$

unde optim = max, dacă $c[j,k]$ reprezintă profitul, şi optim = min, dacă $c[j,k]$ reprezintă costul arcului $\langle j,k\rangle$.

Pe baza relaţiilor 11.2, calculul valorilor ValOpt$[i,j]$ rezultă din rezolvarea următoarelor subprobleme: DODE($V_{p+1}$), DODE($V_p \cup V_{p+1}$), ..., DODE($V_1 \cup \cdots \cup V_{p+1}$) = DODE($V$). Deoarece vârfurile aflate pe un acelaşi etaj sunt numere consecutive, iar

etajele sunt considerate de la stânga la dreapta (de la $s$ spre $t$), rezultă că drumurile optime se vor determina în ordinea descrescătoare a vârfurilor. Aşa cum vom vedea la implementare, valorile $\text{ValOpt}[i, j]$ pot fi memorate într-un tablou unidimensional prin renunţarea la înregistrarea etajului la care aparţine un vârf.

Utilizând 11.2, proprietatea de substructură optimă se caracterizează astfel: un drum optim care pleacă din vârful $j$ conţine drumurile optime care pleacă din orice vârf intermediar al său. Astfel, dacă se cunosc drumurile optime din $\text{DODE}(X)$ atunci se pot determina drumurile optime din $\text{DODE}(X \cup \{j\})$. De aici, determinarea unui drum optim de la $s$ la $t$, după ce se cunosc valorile optime $\text{ValOpt}[i, j]$, se realizează prin determinarea tuturor drumurilor optime $D[i, j]$:

$$\begin{aligned} D[p+1, l] &= () \qquad \text{(drumul vid)} \\ D[i, j] &= (\langle j, k\rangle, D[i+1, k]) \quad \text{dacă} \quad \text{ValOpt}[i, j] = c[j, k] + \text{ValOpt}[i+1, k] \end{aligned}$$

unde un drum este reprezentat printr-o listă de arce. Notăm că drumurile pot fi determinate simultan cu valorile optime.

Aplicarea programării dinamice pentru determinarea drumurilor optime într-un digraf etajat este prezentată schematic de următorul algoritm:

```
procedure alocRes(V, E, c, ValOpt, D)
   ValOpt[p+1,n] ← 0
   D[p+1,n] ← listaVida()
   for i ← p downto 1 do
       for orice j ∈ Vi do
           determină k0 a.î.  c[j,k0] + ValOpt[i+1,k0] =
                optim{c[j,k] + ValOpt[i+1,k] | ⟨j,k⟩ ∈ A}
           ValOpt[i,j] ← c[j,k0]+ValOpt[i+1,k0]
           insereaza(D[i,j], 1, ⟨j,k⟩)
end
```

**Exerciţiul 11.3.1.** Să se descrie un algoritm pentru determinarea drumului optim într-un digraf etajat, bazat pe paradigma programării dinamice, considerând ca stare $\text{DODE}'(X)$ = problema determinării drumurilor optime de la $s$ la orice $j \in X$.

### 11.3.3 Implementare

Pentru implementare, considerăm digraful $G$ reprezentat prin listele de adiacenţă spre interior. Pentru reprezentarea funcţiei ValOpt este suficient să utilizăm un tablou unidimensional prin renunţarea la înregistrarea etajului la care aparţine vârful $j$, *i.e.*, $\text{ValOpt}[j] = \text{ValOpt}[i, j]$. Datorită proprietăţii de substructură optimă, nu este eficient să înregistrăm tot drumul optim care pleacă din vârful $j$. Dacă ştim primul arc $\langle j, k\rangle$ de pe drumul optim, atunci parcurgem acest arc şi apoi continuăm (recursiv) pe drumul optim care pleacă din $k$. În acest mod se parcurge tot drumul optim care pleacă din $j$. Vom utiliza un tablou $S[j]$ care va înregistra numai primul vârf (*Secundul*) de pe drumul optim care pleacă din $j$, iar întregul drum va fi $(j, S[j], S[S[j]], \ldots, n)$ (reprezentat ca listă de vârfuri). Determinarea funcţiei ValOpt presupune parcurgerea digrafului pe etaje, începând cu ultimul

etaj $V_{p-1} = n$. Considerăm numai cazul optim = max. Pentru o descriere uniformă a algoritmului, se presupune că inițial are loc:

$$\text{ValOpt}[j] = -\infty \quad , j = 0, \ldots, n-2$$

unde $-\infty$ va fi reprezentat de o constantă MinusInf, mai mică decât orice $c[i,j]$. Plecând de la ipoteza că pentru vârful $k$ de pe nivelul $i$ valoarea optimă ValOpt[$k$] a fost determinată, se actualizează ValOpt[$j$] pentru toate vârfurile $j$ de pe nivelul $i-1$ cu proprietatea că există arcul $\langle j,k\rangle$. Actualizarea constă în verificarea condiției ValOpt[$j$] < $c[j,k]$ + ValOpt[$k$] și în cazul când este adevărată, noua valoare a lui ValOpt[$j$] este $c[j,k]$ + ValOpt[$k$] iar valoarea variabilei $D[j]$ este $k$. Valoarea $c[j,k]$ este memorată de variabila q->c, unde $q\text{->}varf = j$ iar q este o variabilă referință (pointer) cu care se parcurge lista vârfurilor incidente spre interior în $k$. După completarea tablourilor ValOpt și $S$, se determină drumul optim. Valoarea drumului optim este ValOpt[0].

```
function alocRes(G, ValOpt, D)
   for j ← 0 to G.n-2 do
       ValOpt[j] ← -∞
   ValOpt[G.n-1] ← 0
   for k ← G.n-1 downto 1 do
       q ← G.a[k]
       while (q ≠ NULL) do
          j ← q->varf
          if (ValOpt[j] < ValOpt[k] + q->c)
             then ValOpt[j] ← ValOpt[k] + q->c
                  S[j] ← k
          q ← q->succ
   D[0] ← 0
   D[p] ← n-1
   for i ← 1 to p-1 do
       D[i] ← S[D[i-1]]
   return ValOpt[0]
end
```

**Evaluarea algoritmului** Fiecare arc este cercetat o singură dată, astfel că determinarea celor două tablouri ValOpt și S necesită $O(m)$ timp, unde $m$ este numărul de arce. Deoarece determinarea drumului se face în timpul $\Theta(p)$, rezultă că algoritmul alocRes necesită $O(m)$ timp.

## 11.4 Drumurile cele mai scurte între oricare două vârfuri ale unui digraf

### 11.4.1 Descrierea problemei

Această problemă este o generalizare a problemei determinării drumului optim într-un digraf etajat:

Se consideră $G = (V, A)$ un digraf cu $V = \{0, \ldots, n-1\}$ şi $\ell : A \to \mathcal{R}$ o funcţie de etichetare a arcelor. Perechea $(G, \ell)$ se mai numeşte *digraf ponderat*. Notăm $\ell_{ij}$ în loc de $\ell((i,j))$ şi numim $\ell_{ij}$ *lungimea (= costul) arcului* $(i,j)$. *Lungimea (= costul) unui drum* este suma lungimilor arcelor ce compun drumul. Problema constă în a determina, pentru orice două vârfuri $i, j$, un drum de lungime minimă de la vârful $i$ la vârful $j$ (când există).

## 11.4.2 Modelul matematic

Pentru o prezentare uniformă a metodei, extindem funcţia $\ell$ la $\ell : V \times V \to \mathcal{R}$, punând $\ell_{ij} = \infty$ pentru acele perechi de vârfuri distincte cu $\langle i, j \rangle \notin E$ şi $\ell_{ii} = 0$ pentru orice $i = 0, \ldots, n-1$.

Drept stare definim DM2VD($X$) (Drum Minim între oricare două vârfuri ale unui Digraf) ca fiind subproblema corespunzătoare determinării drumurilor de lungime minimă cu vârfuri intermediare din mulţimea $X \subseteq V$. Evident, DM2VD($V$) este chiar problema iniţială. Notăm cu $\ell_{ij}^X$ lungimea drumului minim de la $i$ la $j$ construit cu vârfuri intermediare din $X$. Dacă $X = \emptyset$, atunci $\ell_{ij}^{\emptyset} = \ell_{ij}$. Considerăm decizia optimă care transformă starea DM2VD($X \cup \{k\}$) în DM2VD($X$). Presupunem că $(G, \ell)$ *este un digraf ponderat fără circuite negative.* Fie $\rho$ un drum optim de la $i$ la $j$ ce conţine vârfuri intermediare din mulţimea $X \cup \{k\}$. Avem $\text{lung}(\rho) = \ell_{ij}^{X \cup \{k\}}$, unde $\text{lung}(\rho)$ este lungimea drumului $\rho$. Dacă vârful $k$ nu aparţine lui $\rho$, atunci politica obţinerii lui $\rho$ corespunde de asemenea şi stării DM2VD($X$) şi, aplicând principiul de optim, obţinem:

$$\ell_{ij}^X = \text{lung}(\rho) = \ell_{ij}^{X \cup \{k\}}$$

În cazul în care $k$ aparţine drumului $\rho$, notăm cu $\rho_1$ subdrumul lui $\rho$ de la $i$ la $k$ şi cu $\rho_2$ subdrumul de la $k$ la $j$. Aceste două subdrumuri au vârfuri intermediare numai din $X$. Conform principiului de optim, politica optimă corespunzătoare stării DM2VD($X$) este subpolitică a politicii optime corespunzătoare stării DM2VD($X \cup \{k\}$). Rezultă că $\rho_1$ şi $\rho_2$ sunt optime în DM2VD($X$). De aici rezultă:

$$\ell_{ij}^{X \cup \{k\}} = \text{lung}(\rho) = \text{lung}(\rho_1) + \text{lung}(\rho_2) = \ell_{ik}^X + \ell_{kj}^X$$

Acum, ecuaţia funcţională analitică pentru valorile optime $\ell_{ij}^X$ are următoarea formă:

$$\ell_{ij}^{X \cup \{k\}} = \min\{\ell_{ij}^X, \ell_{ik}^X + \ell_{kj}^X\} \tag{11.3}$$

**Corolar 11.1.** *Dacă $\langle G, \ell \rangle$ nu are circuite de lungime negativă, atunci au loc următoarele relaţii:*

- $\ell_{kk}^{X \cup \{k\}} = 0$
- $\ell_{ik}^{X \cup \{k\}} = \ell_{ik}^X$
- $\ell_{kj}^{X \cup \{k\}} = \ell_{kj}^X$

*pentru orice $i, j, k \in V$.*

Calculul valorilor optime rezultă din rezolvarea subproblemelor

$$\mathrm{DM2VD}(\emptyset), \mathrm{DM2VD}(\{0\}), \mathrm{DM2VD}(\{0,1\}), \ldots, \mathrm{DM2VD}(\{0,1,\ldots,n-1\}) =$$

$$\mathrm{DM2VD}(V)$$

Convenim să notăm $\ell_{ij}^k$ în loc de $\ell_{ij}^{\{0,\ldots,k-1\}}$. Pe baza corolarului 11.1 rezultă că valorile optime pot fi memorate într-un acelaşi tablou. Maniera de determinare a acestora este asemănătoare cu cea utilizată la determinarea matricei existenţei drumurilor de către algoritmul lui Warshall.

Pe baza ecuaţiilor 11.3, proprietatea de substructură optimă se caracterizează prin: un drum optim de la $i$ la $j$ include drumurile optime de la $i$ la $k$ şi de la $k$ la $j$, pentru orice vârf intermediar $k$ al său. Astfel că drumurile optime din $\mathrm{DM2VD}(X \cup \{k\})$ pot fi determinate utilizând drumurile minime din $\mathrm{DM2VD}(X)$. În continuare considerăm numai cazurile $X = \{0,1,\ldots,k-1\}$. Ca şi în cazul digrafurilor etajate, determinarea drumurilor optime poate fi făcută cu ajutorul unor tablouri $P^k = (P_{ij}^k)$, dar care de această dată au o semnificaţie diferită: $P_{ij}^k$ este vârful penultimului vârf de pe drumul optim de la $i$ la $j$. Pentru $k = 0$ avem $P_{ij}^0 = i$ dacă $\langle i,j\rangle \in E$ şi $P_{ij}^0 = 0$, în celelalte cazuri. Decizia $k$ determină $P^k$ odată cu determinarea matricei $\ell^k = (\ell_{ij}^k)$. Dacă $\ell_{ik}^{k-1} + \ell_{kj}^{k-1} < \ell_{ij}^{k-1}$, atunci drumul optim de la $i$ la $j$ este format din concatenarea drumului optim de la $i$ la $k$ cu drumul optim de la $k$ la $j$ şi deci penultimul vârf de pe drumul de la $i$ la $j$ coincide cu penultimul vârf de pe drumul de la $k$ la $j$: $P_{ij}^k = P_{kj}^{k-1}$. În caz contrar, avem $P_{ij}^k = P_{ij}^{k-1}$. Cu ajutorul matricei $P_{ij}^n$ pot fi determinate drumurile optime: ultimul vârf pe drumul de la $i$ la $j$ este $j_t = j$, penultimul vârf este $j_{t-1} = P_{ij}^n$, antipenultimul este $j_{t-2} = P_{ij_{t-1}}^n$ ş.a.m.d. În acest mod, toate drumurile pot fi memorate utilizând numai $O(n^2)$ spaţiu.

Aplicarea programării dinamice pentru problema determinării drumurilor minime este descrisă de schema procedurală `drmin`:

```
procedure drmin(G, ℓ, P)
   for i ← 0 to n-1 do
      for j ← 0 to n-1 do
```

$$\ell_{ij}^0 = \begin{cases} 0 & , i = j \\ \ell_{ij} & , \langle i,j\rangle \in A \\ \infty & , \text{altfel} \end{cases}$$

$$P_{ij}^0 = \begin{cases} i & , i \neq j, \langle i,j\rangle \in A \\ 0 & , \text{altfel} \end{cases}$$

```
   for k ← 0 to n-1 do
      for i ← 0 to n-1 do
         for j ← 0 to n-1 do
```

$$\ell_{ij}^k = \min\{\ell_{ij}^{k-1}, \ell_{ik}^{k-1} + \ell_{kj}^{k-1}\}$$

$$P_{ij}^k = \begin{cases} P_{ij}^{k-1} & , \ell_{ij}^k = \ell_{ij}^{k-1} \\ P_{kj}^{k-1} & , \ell_{ij}^k = \ell_{ik}^{k-1} + \ell_{kj}^{k-1} \end{cases}$$

```
end
```

### 11.4.3 Implementare

Presupunem că digraful $G = (V, A)$ este reprezentat prin matricea de adiacență, pe care convenim să o notăm aici cu G.L (este ușor de văzut că matricea ponderilor include și reprezentarea lui $A$). Datorită corolarului 11.1, matricele $\ell^k$ și $\ell^{k-1}$ pot fi memorate de același tablou bidimensional G.L. Este ușor de văzut că matricea G.L include și reprezentarea lui $A$. Simbolul $\infty$ este reprezentat de o constantă plusInf cu valoare foarte mare. Dacă digraful are circuite negative, atunci acest lucru poate fi depistat: dacă la un moment dat se obține $G.L[i,i] < 0$, pentru un $i$ oarecare, atunci există un circuit de lungime negativă care trece prin $i$. Funcția drmin întoarce valoarea *true* dacă digraful ponderat reprezentat de matricea G.L nu are circuite negative și în acest caz G.L va conține la ieșire lungimile drumurilor minime între oricare două vârfuri, iar G.P reprezentarea acestor drumuri.

```
procedure drmin(G, P)
   for i ← 0 to n-1 do
      for j ← 0 to n-1 do
         if ((i ≠ j) and (L[i,j] ≠ plusInf))
            then P[i,j] ← i
            else P[i,j] ← 0
   for k ← 0 to n-1 do
      for i ← 0 to n-1 do
         for j ← 1 to n do
            if ((L[i,k] = PlusInf) or (L[k,j] = PlusInf))
               then temp ← plusInf
               else temp ← L[i,k]+L[k,j]
            if (temp < L[i,j])
               then L[i,j] ← temp
                    P[i,j] ← P[k,j]
            if ((i = j) and (L[i,j] < 0))
               then throw '(di)graful are circuite negative'
end
```

**Evaluare.** Se verifică ușor că execuția algoritmului drmin necesită $O(n^3)$ timp și utilizează $O(n^2)$ spațiu.

**Observație.** Algoritmul descris de drmin este cunoscut în literatură sub numele de algoritmul Floyd-Warshall. O formă restrictivă a problemei drumurilor minime se obține prin precizarea vârfului de start. Algoritmii cei mai cunoscuți care rezolvă această formă restrictivă sunt Dijkstra (exercițiul 9.5) și Bellman-Ford (exercițiul 11.2).

sfobs

## 11.5 Problema rucsacului II (varianta discretă)

### 11.5.1 Descrierea problemei

Considerăm următoarea versiune modificată a problemei rucsacului:

Se consideră un rucsac de capacitate $M \in \mathbb{Z}_+$ şi $n$ obiecte $1, \ldots, n$ de dimensiuni (greutăţi) $w_1, \ldots, w_n \in \mathbb{Z}_+$. Un obiect $i$ este introdus în totalitate în rucsac, $x_i = 1$, sau nu este introdus deloc, $x_i = 0$, astfel că o umplere a rucsacului constă dintr-o secvenţă $x_1, \ldots, x_n$ cu $x_i \in \{0, 1\}$ şi $\sum_i x_i \cdot w_i \leq M$. Ca şi în cazul continuu, introducerea obiectului $i$ în rucsac aduce profitul $p_i \in \mathbb{Z}$, iar profitul total este $\sum_{i=1}^{n} x_i p_i$. Problema constă în a determina o alegere $(x_1, \ldots, x_n)$ care să aducă un profit maxim.

Deci, singura deosebire faţă de varianta continuă studiată la metoda *greedy* constă în condiţia $x_i \in \{0, 1\}$, în loc de $x_i \in [0, 1]$.

**Exerciţiul 11.5.1.** Considerăm o variantă mai „veselă" a problemei: un hoţ intră într-un magazin unde sunt $n$ obiecte de dimensiuni şi valori diferite. În sacul hoţului încap numai o parte dintre aceste obiecte. Hoţul trebuie să decidă într-un timp foarte scurt ce obiecte pune în sac astfel încât să aibă un „profit" cât mai mare. Fiind un bun algoritmist, acesta aplică un algoritm *greedy*.

1. Arătaţi că indiferent de criteriul de alegere locală, nu întotdeauna soluţia aleasă este şi cea optimă.
2. Am spus că hoţul era un bun algoritmist. De ce a ales el strategia *greedy* şi nu oricare altă metodă care dă soluţia optimă? (Deşi un răspuns riguros argumentat poate fi dat după citirea acestei secţiuni, încercaţi să-l anticipaţi.)

### 11.5.2 Modelul matematic

Formulată ca problemă de optim, problema rucsacului, varianta discretă, este:

- funcţia obiectiv:

$$\max \sum_{i=1}^{n} x_i \cdot p_i$$

- restricţii:

$$\sum_{i=1}^{n} x_i \cdot w_i \leq M$$
$$x_i \in \{0, 1\}, i = 1, \ldots, n$$
$$w_i \in \mathbb{Z}_+, p_i \in \mathbb{Z}, i = 1, \ldots, n$$
$$M \in \mathbb{Z}$$

O stare este notată cu RUCSAC$(j, X)$ şi reprezintă următoarea problemă, care este o generalizare a celei iniţiale:

- funcţia obiectiv:

$$\max \sum_{i=1}^{j} x_i \cdot p_i$$

- restricţii:

$$\sum_{i=1}^{j} x_i \cdot w_i \leq X$$
$$x_i \in \{0, 1\}, i = 1, \ldots, j$$
$$w_i \in \mathbb{Z}_+, p_i \in \mathbb{Z}, i = 1, \ldots, j$$
$$X \in \mathbb{Z}$$

Cu $f_j(X)$ notăm valoarea optimă pentru instanţa RUCSAC$(j, X)$. Dacă $j = 0$ şi $X \geq 0$, atunci $f_j(X) = 0$. Presupunem $j > 0$. Considerăm decizia optimă prin care starea RUCSAC$(j, X)$ este transformată în RUCSAC$(j-1, ?)$. Notăm cu $(x_1, \ldots, x_j)$ alegerea care dă valoarea optimă $f_j(X)$. Dacă $x_j = 0$ (obiectul $j$ nu este pus în rucsac), atunci, conform principiului de optim, $f_j(X)$ este valoarea optimă pentru starea RUCSAC$(X, j-1)$ şi de aici $f_j(X) = f_{j-1}(X)$. Dacă $x_j = 1$ (obiectul $j$ este pus în rucsac), atunci, din nou conform principiului de optim, $f_j(X)$ este valoarea optimă pentru starea RUCSAC$(j-1, X-w_j)$ şi, de aici, $f_j(X) = f_{j-1}(X-w_j) + p_j$. Combinând relaţiile de mai sus obţinem:

$$f_j(X) = \begin{cases} -\infty, & \text{dacă } X < 0 \\ 0, & \text{dacă } j = 0 \text{ şi } X \geq 0 \\ \max\{f_{j-1}(X), f_{j-1}(X-w_j) + p_j\}, & \text{dacă } j > 0 \text{ şi } X \geq 0 \end{cases} \qquad (11.4)$$

Am considerat $f_j(X) = -\infty$, dacă $X < 0$.

Utilizând 11.4, proprietatea de substructură optimă se caracterizează astfel: soluţia optimă $(x_1, \ldots, x_j)$ a problemei RUCSAC$(j, X)$ include soluţia optimă $(x_1, \ldots, x_{j-1})$ a subproblemei RUCSAC$(j-1, X-x_j w_j)$. Astfel, soluţia optimă pentru RUCSAC$(j, X)$ se poate obţine utilizând soluţiile optime pentru subprobleme RUCSAC$(i, Y)$ cu $1 \leq i < j, 0 \leq Y \leq X$. Notăm că 11.4 implică o recursie în cascadă şi deci numărul de subprobleme de rezolvat este $O(2^n)$. De aceea, în continuare ne ocupăm de calculul şi memorarea eficientă a valorilor optime pentru subprobleme. Mai întâi considerăm următorul exemplu.

**Exemplu.** Fie $M = 10, n = 3$ şi greutăţile şi profiturile date de următorul tabel:

| $i$ | 1 | 2 | 3 |
|---|---|---|---|
| $w_i$ | 3 | 5 | 6 |
| $p_i$ | 10 | 30 | 20 |

Valorile optime pentru subprobleme sunt calculate cu ajutorul relaţiilor 11.4 şi pot fi memorate într-un tablou bidimensional astfel:

| $X$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $f_0$ | 0 | [0] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $f_1$ | 0 | 0 | 0 | 10 | 10 | [10] | 10 | 10 | 10 | 10 | 10 |
| $f_2$ | 0 | 0 | 0 | 10 | 10 | 30 | 30 | 30 | 40 | 40 | [40] |
| $f_3$ | 0 | 0 | 0 | 10 | 10 | 30 | 30 | 30 | 40 | 40 | [40] |

Tabloul de mai sus este calculat linie cu linie: pentru a calcula valorile de pe o linie sunt consultate numai valorile de pe linia precedentă. Algoritmul de calcul este foarte simplu. De exemplu, $f_2(8) = \max\{f_1(8), f_1(8-5)+30\} = \max\{10, 40\} = 40$. Valoarea optimă a funcţiei obiectiv este $f_3(10) = 40$. Soluţia care dă această valoare se determină în modul următor: se testează dacă $f_2(10) = f_3(10)$. Răspunsul este afirmativ şi, de aici, rezultă $x_3 = 0$, i.e. obiectul 3 nu este pus în rucsac. În continuare se testează dacă $f_1(10) = f_2(10)$. De data aceasta, răspunsul este negativ. Rezultă că $f_2(10) = f_1(10-5) + 30 = f_1(5) + 30$ şi, de aici, $x_2 = 1$, i.e., obiectul 2 este pus în rucsac. În continuare se testează dacă $f_0(5) = f_1(5)$. Şi de

data aceasta răspunsul este negativ şi deci $x_1 = 1$. Dinamica valorilor comparate este sugerată în tablou prin scrierea acestora în dreptunghiuri.

Tabloul de mai sus are dimensiunea $n \cdot m$ (au fost ignorate prima linie şi prima coloană). Dacă $m = O(2^n)$ rezultă că atât complexitatea spaţiu, cât şi cea timp sunt exponenţiale. Privind tabloul de mai sus observăm că există multe valori care se repetă. În continuare ne punem problema memorării mai compacte a acestui tablou. Construim graficele funcţiilor $f_0, f_1, f_2$ şi $f_3$ pentru exemplul de mai sus. Avem:

$$f_0(X) = \begin{cases} -\infty & , X < 0 \\ 0 & , X \geq 0 \end{cases}$$

Notăm cu $g_0$ funcţia dată de:

$$g_0(X) = f_0(X - w_1) + p_1 = \begin{cases} -\infty & , X < 3 \\ 10 & , 3 \leq X \end{cases}$$

Graficele funcţiilor $f_0$ şi $g_0$ sunt reprezentate în figura 11.3.

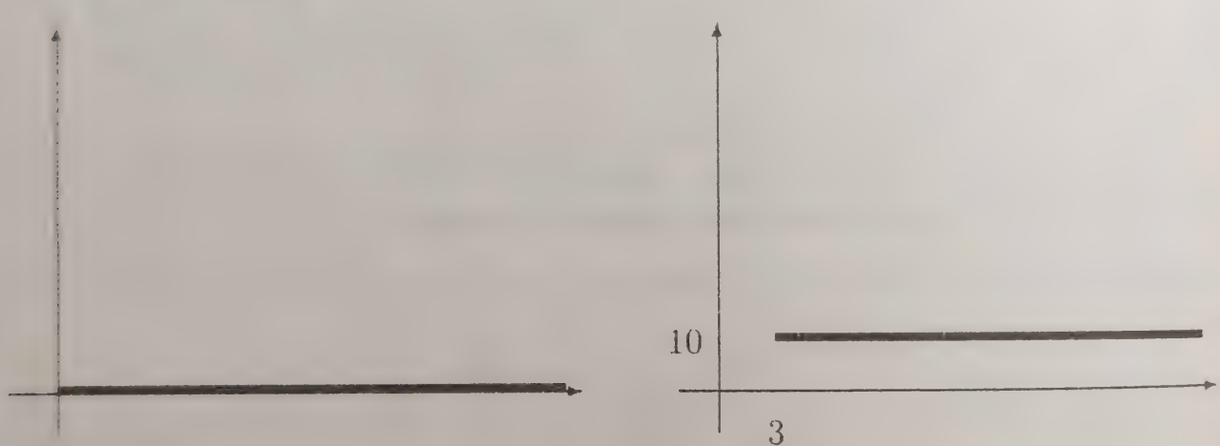



Figura 11.3: Funcţiile $f_0$ şi $g_0$

Funcţia $f_1$ se calculează prin:

$$f_1(X) = \max\{f_0(X), g_0(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X \end{cases}$$

Notăm cu $g_1$ funcţia dată de:

$$g_1(X) = f_1(X - w_2) + p_2 = \begin{cases} -\infty & , X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 \leq X \end{cases}$$

Graficele funcţiilor $f_1$ şi $g_1$ sunt reprezentate în figura 11.4.

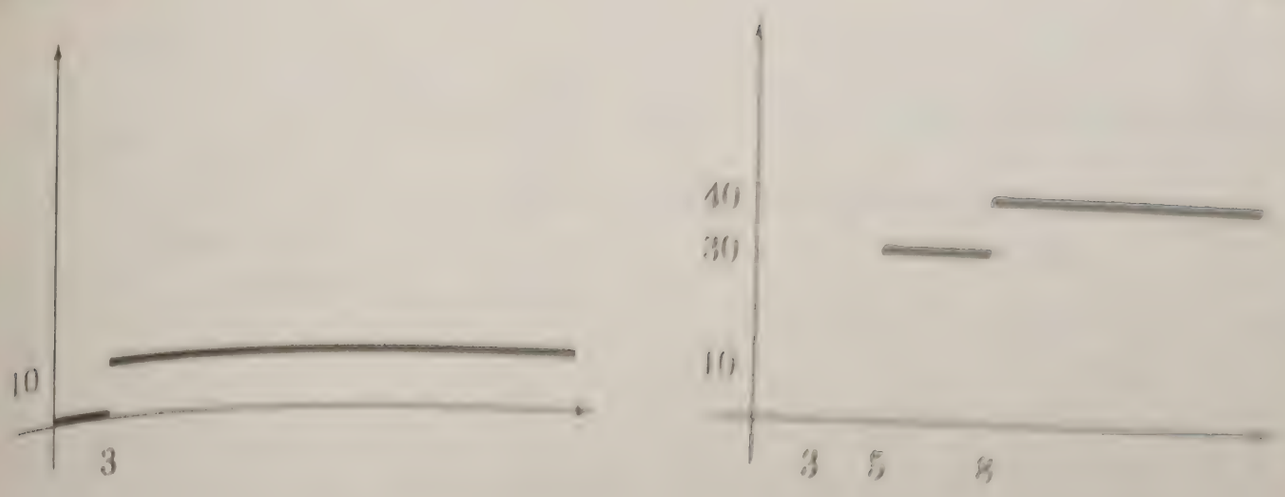



Figura 11.4: Funcţiile $f_1$ şi $g_1$

Funcţia $f_2$ se calculează prin:

$$f_2(X) = \max\{f_1(X), g_1(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 \leq X \end{cases}$$

În continuare, notăm cu $g_2$ funcţia dată prin:

$$g_2(X) = f_2(X - w_3) + p_3 = \begin{cases} -\infty & , X < 6 \\ 20 & , 6 \leq X < 9 \\ 30 & , 9 \leq X < 11 \\ 50 & , 11 \leq X < 14 \\ 60 & , 14 \leq X \end{cases}$$

Graficele funcţiilor $f_2$ şi $g_2$ sunt reprezentate în figura 11.5.
Funcţia $f_3$ se calculează prin:

$$f_3(X) = \max\{f_2(X), g_2(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 < X \leq 11 \\ 50 & , 11 \leq X < 14 \\ 60 & , 14 \leq X \end{cases}$$

Graficul funcţiei $f_3$ este reprezentat în figura 11.6.
Se remarcă faptul că funcţiile $f_i$ şi $g_i$ sunt funcţii în scară. Graficele acestor funcţii pot fi reprezentate prin mulţimi finite din puncte din plan. De exemplu,

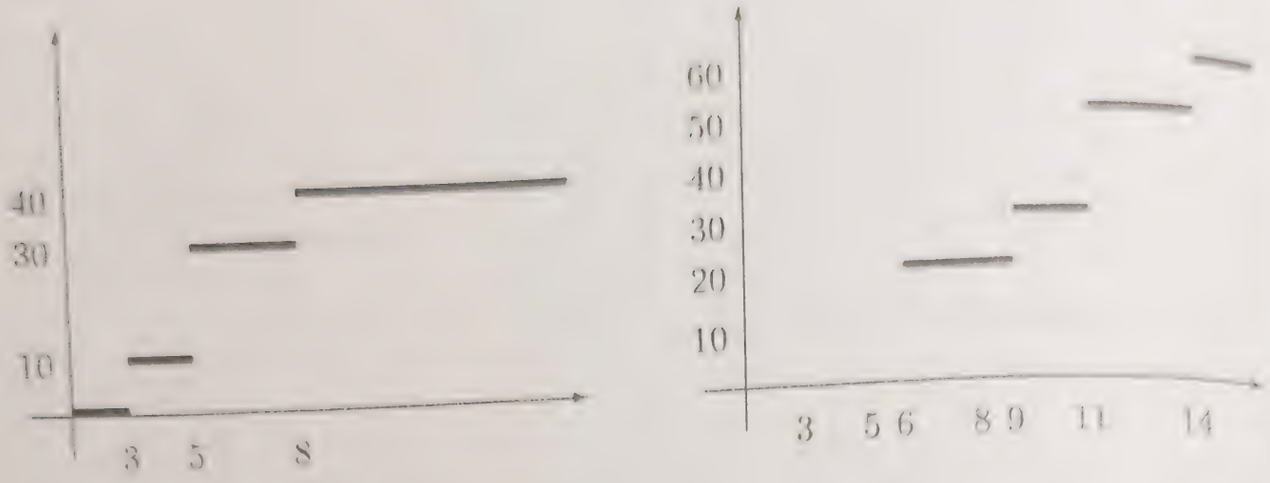



Figura 11.5: Funcţiile $f_2$ şi $g_2$

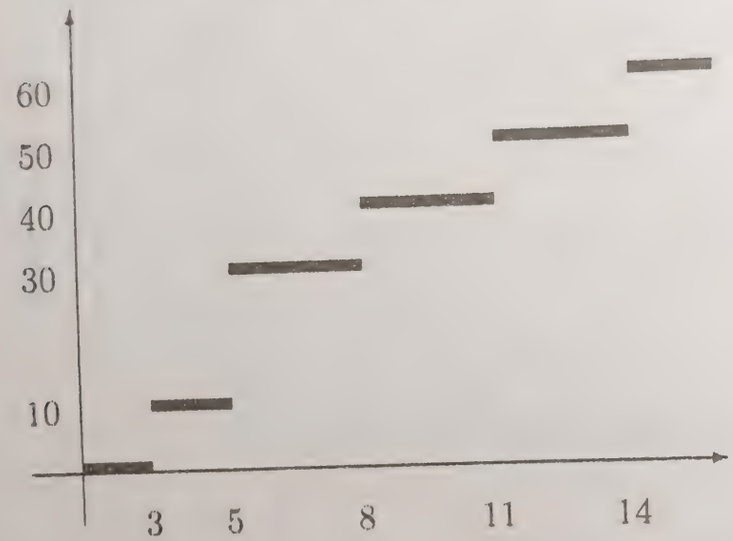



Figura 11.6: Funcţia $f_3$

graficul funcţiei $f_2$ este reprezentat prin mulţimea $\{(0,0),(3,10),(5,30),(8,40)\}$. O mulţime care reprezintă o funcţie în scară conţine acele puncte în care funcţia face salturi. Graficul funcţiei $g_i$ se obţine din graficul funcţiei $f_i$ printr-o translaţie, iar graficul funcţiei $f_{i+1}$ se obţine prin interclasarea graficelor funcţiilor $f_i$ şi $g_i$. sfex

În general, fiecare $f_i$ este complet specificat de o mulţime $S_i = \{(X_j, Y_j) \mid j = 0,\ldots,r\}$, unde $Y_j = f_i(X_j)$. Presupunem $X_1 < \cdots < X_r$. Analog, funcţiile $g_i$ sunt reprezentate prin mulţimile $T_i = \{(X+w_i, Y+p_i) \mid (X,Y) \in S_i\}$. Notăm $T_i = \tau(S_i)$ şi $S_{i+1} = \mu(S_i, T_i)$. Mulţimea $S_{i+1}$ se obţine din $S_i$ şi $T_i$ prin interclasare. Operaţia de interclasare se realizează într-un mod asemănător cu cel de la interclasarea a două linii ale orizontului. Se consideră o variabilă $L$ care ia valoarea 1 dacă graficul lui $f_{i+1}$ coincide cu cel al lui $f_i$ şi cu 2 dacă el coincide cu cel al lui $g_i$. Deoarece $(0,0)$ aparţine graficului rezultat, considerăm $L = 1, j = 1$ şi $k = 1$. Presupunând că la un pas al interclasării se compară $(X_j, Y_j) \in S_i$ cu $(X_k, Y_k) \in T_i$, atunci:

- dacă $L = 1$:
  - dacă $X_j < X_k$, atunci se adaugă $(X_j, Y_j)$ în $S_{i+1}$ şi se incrementează $j$;

- dacă $X_j = X_k$:
  - * dacă $Y_j > Y_k$, atunci se adaugă $(X_j, Y_j)$ în $S_{i+1}$ şi se incrementează $j$ şi $k$;
  - * dacă $Y_j < Y_k$, atunci se adaugă $(X_k, Y_k)$ în $S_{i+1}$, $L = 2$ şi se incrementează $j$ şi $k$;
- dacă $X_j > X_k$, atunci, dacă $Y_k > Y_j$, se adaugă $(X_k, Y_k)$ în $S_{i+1}$, $L = 2$ şi se incrementează $k$;

• dacă $L = 2$:

- dacă $X_j < X_k$, atunci, dacă $Y_j > Y_k$, se adaugă $(X_j, Y_j)$ în $S_{i+1}$, $L = 1$ şi se incrementează $j$;
- dacă $X_j = X_k$:
  - * dacă $Y_j < Y_k$, atunci se adaugă $(X_k, Y_k)$ în $S_{i+1}$ şi se incrementează $j$ şi $k$;
  - * dacă $Y_j > Y_k$, atunci se adaugă $(X_j, Y_j)$ în $S_{i+1}$, $L = 1$ şi se incrementează $j$ şi $k$;
- dacă $X_j > X_k$, atunci se adaugă $(X_k, Y_k)$ în $S_{i+1}$ şi se incrementează $k$;

Notăm cu `interclGrafice(S_i, T_i)` funcţia care determină $S_{i+1}$ conform algoritmului de mai sus. Rămâne de extras soluţia optimă din $S_n$. Considerăm mai întâi cazul din exemplul de mai sus.

**Exemplu** (continuare).

• Se caută în $S_n = S_3$ perechea $(X_j, Y_j)$ cu cel mai mare $X_j$ pentru care $X_j \leq M$. Obţinem $(X_j, Y_j) = (8, 40)$. Deoarece $(8, 40) \in S_3$ şi $(8, 40) \in S_2$ rezultă $f_{optim}(M) = f_{optim}(8) = f_3(8) = f_2(8)$ şi deci $x_3 = 0$. Perechea $(X_j, Y_j)$ rămâne neschimbată.

• Pentru că $(X_j, Y_j) = (8, 40)$ este în $S_2$ şi nu este în $S_1$, rezultă că $f_{optim}(8) = f_1(8-w_2)+p_2$ şi deci $x_2 = 1$. În continuare se ia $(X_j, Y_j) = (X_j - w_2, Y_j - p_2) = (8 - 5, 40 - 30) = (3, 10)$.

• Pentru că $(X_j, Y_j) = (3, 10)$ este în $S_1$ şi nu este în $S_0$, rezultă că $f_{optim}(3) = f_1(3 - w_1) + p_1$ şi deci $x_1 = 1$.

Metoda poate fi descrisă pentru cazul general:

• Iniţial se determină perechea $(X_j, Y_j) \in S_n$ cu cel mai mare $X_j$ pentru care $X_j \leq M$. Valoarea $Y_j$ constituie încărcarea optimă a rucsacului, *i.e.*, valoarea funcţiei obiectiv din problema iniţială.

• Pentru $i = n - 1, \ldots, 0$:

- dacă $(X_j, Y_j)$ este în $S_i$, atunci $f_{i+1}(X_j) = f_i(X_j) = Y_j$ şi se face $x_{i+1} = 0$ (obiectul $i + 1$ nu este ales);
- dacă $(X_j, Y_j)$ nu este în $S_i$, atunci $f_{i+1}(X_j) = f_i(X_j - w_{i+1}) + p_{i+1} = Y_j$ şi se face $x_{i+1} = 1$ (obiectul $i+1$ este ales), $X_j = X_j - w_{i+1}$ şi $Y_j = Y_j - p_{i+1}$.

### 11.5.3 Implementare

Descrierea algoritmică completă a metodei prezentate anterior este următoarea:

```
procedure rucsac_II(n, w, p, val0pt, x)
    S0 ← {(0,0)}
    T0 ← {(w1,p1)}
    for i ← 1 to n
        Si(X) ← interclGrafice(Si-1,Ti-1)
        Ti ← {(X + wi,Y + pi) | (X,Y) ∈ Si}
    determină (Xj,Yj) cu Xj = max{Xi | (Xi,Yi) ∈ Sn,Xi ≤ M}
    for i ← n-1 downto 1 do
        if (Xj,Yj) ∈ Si
           then xi+1 ← 0
           else xi+1 ← 1
                Xj ← Xj - wi+1
                Yj ← Yj - pi+1
end
```

**Evaluare.** Ne propunem să determinăm timpul de execuţie şi spaţiul utilizat de algoritmul rucsac_II. Notăm $m = \sum_{i=0}^{n} \#S_i$. Deoarece $\#T_i = \#S_i$, rezultă că $\#S_{i+1} \leq 2 \cdot \#S_i$ şi de aici $\sum_i \#S_i \leq \sum_i 2^i = 2^n - 1$. Calculul lui $S_i$ din $S_{i-1}$ necesită timpul $\Theta(\#S_{i-1})$ şi de aici calculul lui $S_n$ necesită timpul $\sum_i \Theta(\#S_i) = O(T)$. Deoarece profiturile $p_i$ sunt numere întregi, pentru orice $(X,Y) \in S_i$, $Y$ este întreg şi $Y \leq \sum_{j \leq i} p_j$. Analog, pentru că dimensiunile $w_i$ sunt numere întregi, pentru $(X,Y) \in S_i$, $X$ este întreg şi $X \leq \sum_{j \leq i} w_j$. Deoarece perechile $(X,Y)$ cu $X > M$ nu interesează, ele pot să nu fie incluse în mulţimile $S_i$. De aici, rezultă că numărul maxim de perechi $(X,Y)$ distincte din $S_i$ satisface relaţiile:

$$\#S_i \leq 1 + \sum_{j=1}^{i} w_j \quad \text{şi} \quad \#S_i \leq M$$

care implică

$$\#S_i \leq 1 + \min\{\sum_{j=1}^{i} w_j, M\}.$$

Relaţia de mai sus permite o estimare mai precisă a spaţiului necesar pentru memorarea mulţimilor $S_i$ în cazul unor probleme concrete. În ceea ce priveşte timpul, făcând calculele, rezultă că algoritmul are complexitatea timp

$$O(\min(2^n, n\sum_{i=1}^{n} p_i, nM)).$$

# 11.6 Subsecvenţa crescătoare maximală

## 11.6.1 Descrierea problemei

Fie $a = (a_1, \ldots, a_n)$ o secvenţă de numere întregi. Ştergând câteva elemente din $a$ se obţine o nouă secvenţă, pe care o numim *subsecvenţă*. O subsecvenţă se numeşte *crescătoare* dacă elementele sale sunt în ordine crescătoare. De exemplu, dacă $a = (7, 3, 8, 5, 13)$, atunci subsecvenţa $(7, 3, 5)$ nu este crescătoare, iar subsecvenţa $(7, 8, 13)$ este crescătoare. Problema constă în determinarea unei subsecvenţe crescătoare de lungime maximă. Dacă există mai multe, atunci se determină una arbitrară dintre ele.

## 11.6.2 Modelul matematic

Se ataşează problemei un digraf $G = (V, E)$, unde

$$V = \{0, 1, \ldots, n\}$$
$$A = \{(i, j) \mid a[i] \leq a[j], 1 \leq i < j \leq n\} \cup \{\langle 0, i\rangle \mid i = 1, \ldots, n\}$$

Determinarea unei subsecvenţe crescătoare maximale este echivalentă cu determinarea unui drum de lungime maximă în $G$.

**Exemplu.** Digraful asociat secvenţei $a = (3, 1, 4, 6, 3)$ este cel reprezentat în figura 11.7.

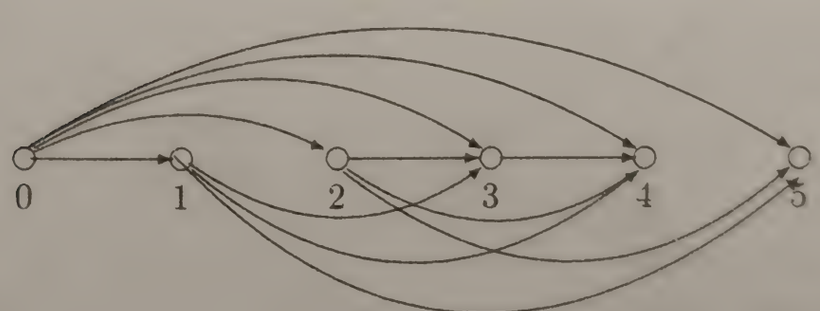



Figura 11.7: Digraful asociat secvenţei $a = (3, 1, 4, 6, 3)$

sfex

Se consideră următoarea matrice de costuri pe arce (ponderi):

$$c[i, j] = \begin{cases} 1 & \text{, dacă } (i < j \text{ şi } a_i \leq a_j) \text{ sau } (i = 0), \\ -\infty & \text{, altfel.} \end{cases}$$

Notăm cu SCM($i$) subproblema determinării lungimii celui mai lung drum care se termină în vârful $i$, echivalent, subproblema determinării subsecvenţei crescătoare

maximale ce are pe ultimul loc pe $a[i]$. Fie $L[i]$ valoarea optimă pentru SCM($i$). Dacă $i = 0$, atunci $L[0] = 0$. Presupunem $i > 0$. Fie $j$ sursa ultimului arc de pe drumul care dă valoarea optimă $L[i]$. Evident, $j < i$. Aplicând principiul de optim, politica optimă corespunzătoare stării SCM'($j$) este subpolitică a politicii optime corespunzătoare stării SCM'($i$). Rezultă $L[i] = L[j] + c[j, i]$. Deci $L[i]$ satisface următoarea relaţie de recurenţă:

$$L[i] = \begin{cases} 0 & \text{, dacă } i = 0, \\ \max_{j<i} L[j] + c[j, i] & \text{, dacă } i > 0. \end{cases} \tag{11.5}$$

Pentru a determina o secvenţă maximală din tabelul valorilor optime, se procedează într-un mod asemănător cu determinarea drumului optim din matricea ce memorează penultimul vârf de pe fiecare drum optim. Aici nu este nevoie de memorarea penultimului vârf, pentru că el poate fi determinat din tabelul valorilor optime. Se procedează în modul următor:

- se determină $j$ cu $L[j] = \max_i L[i]$;
- $k \leftarrow L[j]$; $s[k] \leftarrow a[j]$;
- următorul proces se repetă până când $k$ devine 1:
  - se determină $j_1$ cu $j_1 < j$, $a[j_1] \leq a[j]$ şi $L[j_1] = L[j] - 1$;
  - $k \leftarrow k - 1$; $j \leftarrow j_1$; $s[k] \leftarrow a[j]$.

### 11.6.3 Implementare

Pentru implementare nu este necesară construirea efectivă a digrafului. Acesta a fost utilizat numai pentru găsirea şi verificarea corectitudinii algoritmului.

```
procedure scm(a, n, sm, lg)
   L[1] ← 1
   for i ← 2 to n do
       k ← 0
       for j ← 1 to i-1 do
           if ((a[j] ≤ a[i]) and (L[j] > k))
              then k ← L[j]
       L[i] ← k+1
       lg ← L[1]
       j ← 1
       for i ← 2 to n do
           if (L[i] > lg)
              then lg ← L[i]
                   j ← i
       sm[lg] ← a[j]
       for i ← lg-1 downto 1 do
           k ← j-1
           while ((L[k]+1 ≠ L[j]) or (a[k] > a[j])) do
              k ← k-1
           sm[i] ← a[k]
           j ← k
end
```

**Evaluare.** Prima parte, care determină lungimile subsecvenţelor crescătoare maximale care se termină în $i$, pentru $i = 1, \ldots, n$, se realizează în timpul $O(n^2)$. A doua etapă, care determină secvenţa optimă utilizând tabelul lungimilor optime, necesită timpul $O(n)$. Rezultă că întreg algoritmul are complexitatea timp $O(n^2)$. Memorarea secvenţei iniţiale, subsecvenţei crescătoare maximale şi a lungimilor utilizează $O(n)$ spaţiu.

**Observaţie.** Performanţa timp a programului `DetSCM2` poate fi îmbunătăţită prin utilizarea suplimentară a unui tablou p şi completând acest tablou în prima parte punând $p[j] = k$, unde $k$ dă maximul. Acest tablou poate fi utilizat la determinarea subsecvenţei crescătoare maximale în timpul $O(n)$. Utilizarea tabloului p este similară cu cea a tabloului bidimensional p de la drumurile minime. sfobs

## 11.7 Distanţa între şiruri

### 11.7.1 Descrierea problemei

Se consideră două şiruri $\alpha = a_1 \cdots a_n$ şi $\beta = b_1 \cdots b_n$ formate cu litere dintr-un alfabet $A$. Asupra şirului $\alpha$ se pot face următoarele operaţii:

- *Ştergere*: $S(i)$ şterge litera de pe poziţia $i$;
- *Inserare*: $I(i, c)$ inserează litera $c$ pe poziţia $i$;
- *Modificare*: $M(i, c)$ înlocuieşte litera de pe poziţia $i$ cu $c$.

Problema constă în determinarea unei secvenţe de operaţii de lungime minimă care transformă pe $\alpha$ în $\beta$.

**Exemplu.** Fie $\alpha = carnet, \beta = paleta$. O secvenţă de transformări este $carnet \mapsto parnet \mapsto palnet \mapsto palet \mapsto paleta$. Transformările efectuate sunt $M(1, p), M(3, l), S(4)$ şi $I(6, a)$. Există o altă secvenţă mai scurtă? sfex

### 11.7.2 Modelul matematic

Mai întâi dovedim câteva proprietăţi ale secvenţelor optime de transformări care duc pe $\alpha$ în $\beta$.

**Lema 11.1.** *Fie $s = (\ldots T(i, \bullet), T'(j, \bullet) \ldots)$ o secvenţă optimă (de lungime minimă) care transformă pe $\alpha$ în $\beta$. Atunci există $k, \ell$, astfel încât secvenţa $s' = (\ldots T'(k, \bullet), T(\ell, \bullet) \ldots)$, obţinută din $s$ prin interschimbarea celor două operaţii $T$ şi $T'$ şi în rest rămânând neschimbată, este de asemenea o secvenţă optimă care transformă pe $\alpha$ în $\beta$.*

*Demonstraţie.* Distingem următoarele cazuri.
1. $T(i, \bullet) = S(i), T'(j, \bullet) = S(j)$. Nu contează ordinea în care sunt şterse două litere. $k$ şi $\ell$ sunt poziţiile de ştergere actualizate.
2. $T(i, \bullet) = S(i), T'(j, \bullet) = I(j, c)$. Dacă $i > j$, atunci $T'(k, \bullet) = I(j, c)$ şi $T(\ell, \bullet) = S(i + 1)$. Dacă $i \leq j$, atunci $T'(k, \bullet) = I(j + 1, c)$ şi $T(\ell, \bullet) = S(i)$.
3. $T(i, \bullet) = S(i), T'(j, \bullet) = M(j, c)$. Analog cazului 2.

4. $T(i,\bullet) = I(i,c), T'(j,\bullet) = S(j)$. Dacă $i \neq j$, atunci se procedează ca la 2. Dacă $i = j$, atunci, datorită faptului că execuţia succesivă a operaţiilor $I(i,c)$, $S(i)$ lasă neschimbat şirul, rezultă că secvenţa de operaţii nu ar fi optimă.
5. $T(i,\bullet) = I(i,c), T'(j,\bullet) = M(j,c)$. Analog cu 2.
6. $T(i,\bullet) = I(i,c), T'(j,\bullet) = I(j,c)$. Nu are importanţă ordinea în care sunt inserate două litere. $k$ şi $\ell$ sunt poziţiile de inserare actualizate.

Celelalte cazuri, când $T(i,\bullet) = M(i,c)$ se tratează în mod asemănător. sfdem

**Corolar 11.2.** *Există o secvenţă optimă care dacă efectuează transformările:*

$$\alpha = \alpha_0 \mapsto \alpha_1 \mapsto \cdots \mapsto \alpha_t = \beta$$

*atunci pentru orice $i$, $|\alpha_i| \leq |\beta|$, unde prin $|\_|$ am notat lungimea şirului $\_$.*

*Demonstraţie.* Primele transformări care se fac sunt ştergerile. sfdem

Notăm cu $d(\alpha,\beta)$ lungimea unei secvenţe optime.

**Lema 11.2.** *Are loc:*

*i)* $d(\alpha,\alpha) = 0$;
*ii)* $d(\alpha,\beta) = d(\beta,\alpha)$;
*iii)* $d(\alpha,\gamma) \leq d(\alpha,\beta) + d(\beta,\gamma)$.

*Demonstraţie.* i) este evidentă. Proprietatea ii) rezultă din faptul că orice operaţie $T : \alpha \to \beta$ are inversă $T^{-1} : \beta \to \alpha$:

$$S^{-1}(i) = I(i,a_i)$$
$$I^{-1}(i,c) = S(i)$$
$$M^{-1}(i,c) = M(i,a_i)$$

iii) rezultă din condiţia de minim. sfdem

**Observaţie.** Lema de mai sus arată că $d(\alpha,\beta)$ este o metrică. De aici şi numele problemei. Această metrică mai este numită şi *distanţa Levenshtein*, după numele matematicianului rus care a proiectat algoritmul în 1965. sfobs

În continuare aplicăm metoda programării dinamice pentru determinarea secvenţei optime. Definim drept stare subproblema $DS(\alpha_i,\beta_j)$ corespunzătoare transformării subşirului $\alpha_i = a_1 \ldots a_i$ în $\beta_j = b_1 \ldots b_j$ şi prin $d[i,j]$ valoarea optimă $d(\alpha_i,\beta_j)$. Dacă $i = 0$, $\alpha_0$ este şirul vid şi $\beta_j$ se obţine prin $j$ inserări: deci $d[0,j] = j$. Dacă $j = 0$, atunci $\beta_j$ se obţine prin $i$ ştergeri şi avem $d[i,0] = i$. Presupunem $i,j > 0$. Considerăm decizia optimă prin care starea $DS(a_1 \ldots a_i, b_1 \ldots b_j)$ este transformată într-o stare $DS(a_1 \ldots a_{i'}, b_1 \ldots b_{j'})$ cu ($i' < i$ şi $j' \leq j$) sau ($i' \leq i$ şi $j' < j$).

Distingem următoarele situaţii:

1. Dacă $a_i = b_j$, atunci $i' = i-1, j' = j-1$ şi, aplicând principiul de optim, obţinem $d[i,j] = d[i-1,j-1]$.
2. $DS(a_1,\ldots,a_i,b_1,\ldots,b_j)$ se obţine prin ştergere. Rezultă $i' = i-1, j' = j$ şi, aplicând principiul de optim, obţinem $d[i,j] = d[i-1,j] + 1$.

3. $DS(a_1, \ldots, a_i, b_1, \ldots, b_j)$ se obţine prin inserare. Avem $i' = i, j' = j - 1$ şi, aplicând principiul de optim, obţinem $d[i, j] = d[i, j-1] + 1$. Din corolarul lemei precedente rezultă că această operaţie poate fi realizată numai dacă $i < j$.
4. $DS(a_1, \ldots, a_i, b_1, \ldots, b_j)$ se obţine prin modificare. Avem $i' = i - 1, j' = j - 1$ şi, aplicând principiul de optim, obţinem $d[i, j] = d[i-1, j-1] + 1$.

Deoarece $d[i, j]$ trebuie să fie minimă, rezultă:

$$d[i, j] = \min\{d[i-1, j] + 1, d[i-1, j-1] + \delta, d[i, j-1] + 1\}$$

unde

$$\delta = \begin{cases} 0 & \text{, dacă } a_i = b_j \\ 1 & \text{, dacă } a_i \neq b_j \end{cases}$$

În continuare vom utiliza valorile $d[i, j]$ la determinarea secvenţei optime. Procedeul este asemănător cu cel de la subsecvenţa maximală, numai că acum se caută într-un tablou bidimensional. Notăm cu $L$ lista care înregistrează transformările soluţiei optime. Se procedează în modul următor:

- iniţial se consideră $i \leftarrow n$; $j \leftarrow n$; şi $L \leftarrow ( )$ (lista vidă);
- următorul proces se repetă până când $i$ şi $j$ devin 0:
  - dacă $d[i, j] = d[i-1, j-1]$, atunci $L$ rămâne neschimbată şi se face i $\leftarrow i - 1$; j $\leftarrow j - 1$;
  - altfel, dacă $d[i, j] = d[i-1, j-1] + 1$, atunci se face $L \leftarrow M(i, b_j)@L$ şi i $\leftarrow i - 1$; j $\leftarrow j - 1$;
  - altfel, dacă $d[i, j] = d[i-1, j] + 1$, atunci se face $L \leftarrow S(i)@L$ şi i $\leftarrow i - 1$;
  - altfel, dacă $d[i, j] = d[i, j-1] + 1$, atunci se face $L \leftarrow I(i, b_j)@L$ şi j $\leftarrow j - 1$.

Procedura care determină distanţele este:

```
procedure distSir(a, b, n)
   for j ← 1 to n do
       d[0,j] ← j
   for i ← 1 to n do
       d[i,0] ← i
   for i ← 1 to n do
       for j ← 1 to n do
           δ ← if (a[i]=b[j]) then 0 else 1
           d[i,j] ← min{d[i-1,j] + 1, d[i-1,j-1] + δ, d[i,j-1] + 1}
   L ← listaVida()
   i ← n
   j ← n
   repeat
      if (d[i,j] = d[i-1,j-1])
        then i ← i-1
             j ← j-1
        else if (d[i,j] = d[i-1,j-1]+1)
```

```
                then adLaInc(L, ('M', i, b[j]))
                     i ← i-1
                     j ← j-1
                else if (d[i,j]=d[i-1,j]+1)
                        then adLaInc(L, ('S', i))
                             i ← i-1
                        else adLaInc(L, ('I', i, b[j]))
                             j ← j-1
        until ((i = 0) and (j = 0))
    end
```

### 11.7.3 Implementare

Valorile $d[i, j]$ sunt memorate într-un tablou bidimensional şi ordinea de calcul este linie cu linie, începând cu linia 1. Secvenţa de transformări va fi memorată într-o listă liniară abstractă.

Notăm că utilizarea programului de mai sus necesită o implementare a tipului abstract Lista.

**Evaluarea** algoritmului este simplă şi conduce la următorul rezultat:

**Teorema 11.1.** *Determinarea distanţei optime şi a secvenţei optime care transformă un şir $\alpha$ într-un şir $\beta$, $|\alpha| = |\beta| = n$, se poate face în timpul $O(n^2)$.*

## 11.8 Arbori binari de căutare optimali

### 11.8.1 Problema arborelui binar de căutare optimal

Fie $A = (a_1, a_2, \ldots, a_n)$ o secvenţă de chei sortată crescător din care se construieşte un arbore binar de căutare. Se consideră că operaţia de căutare se execută cu anumite frecvenţe. Anume:

Notăm cu $p_i$ probabilitatea de a fi căutat elementul $a_i, i = 1, \ldots, n$ şi cu $P$ secvenţa $(p_1, p_2, \ldots, p_n)$. De asemenea, notăm cu $q_i$ probabilitatea de a fi căutat un element $x$ cu proprietatea $a_i < x < a_{i+1}, i = 0, \ldots, n$, unde am presupus de = $-\infty, a_{n+1} = +\infty$. Secvenţa $(q_0, q_1, \ldots, q_n)$ va fi notată cu $Q$.

În aceste condiţii avem:

$\sum_{i=1}^{n} p_i$ – probabilitatea căutărilor terminate cu succes;

$\sum_{i=0}^{n} q_i$ – probabilitatea căutărilor terminate fără succes

şi

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

Pentru $P$ şi $Q$ date, *arborele binar de căutare optimal* este cel pentru care operaţia de căutare oferă timp mediu minim.

### 11.8.2 Modelul matematic

Pentru a pune în evidenţă timpul mediu se consideră arborele binar de căutare completat cu o serie de noduri pendante corespunzătoare intervalelor de insucces. În figura 11.8, se prezintă un arbore binar de căutare şi arborele completat cu nodurile pendante.

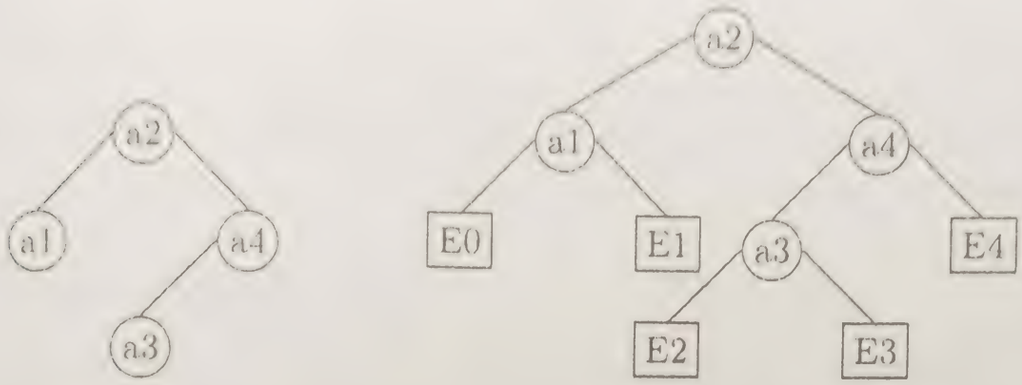



Figura 11.8: Arbore binar de căutare completat cu noduri pendante

Nodul $E_i$ corespunde tuturor căutărilor fără succes pentru valori cuprinse în intervalul $(a_i, a_{i+1})$. Timpul unei operaţii de căutare pentru un element $x$ este dat de adâncimea nodului de valoare $x$ sau de adâncimea pseudonodului corespunzător intervalului care-l conţine pe $x$.

Costul unui arbore binar de căutare se defineşte ca timpul mediu al unei căutări pentru o valoare $x$, adică:

$$\text{cost}(T) = \sum_{i=1}^{n} p_i * \text{nivel}(a_i) + \sum_{i=0}^{n} q_i * (\text{nivel}(E_i) - 1) \qquad (11.8.1)$$

Prin `nivel`$(a)$ se notează nivelul nodului $a$ şi reprezintă numărul de comparaţii efectuate de funcţia de căutare pe drumul de la rădăcină până la nodul $a$. Ponderând acest număr cu probabilitatea de a fi căutat nodul $a$ şi efectuând suma pentru toate nodurile se obţine timpul mediu de căutare cu succes a unui nod. În mod similar, se calculează timpul mediu de căutare pentru insucces, cu observaţia că din `nivel`$(E_i)$ se scade valoarea 1, deoarece decizia de apartenenţă la un interval nu se face la nivelul pseudonodului, ci la nivelul părintelui său.

Dată fiind secvenţa de valori $A$, se pot construi o mulţime de arbori binari de căutare cu cheile nodurilor din $A$. Un arbore binar de căutare optimal este arborele de cost minim. Construcţia unui astfel de arbore are la bază modelul programării dinamice. Programarea dinamică reduce numărul de încercări eliminând o serie de secvenţe care nu pot fi optimale şi aceasta apelând la principiul optimalităţii:

Din punctul de vedere al programării dinamice, construirea unui arbore constă într-un şir de decizii privind nodul care se alege ca rădăcină la fiecare pas. Fie $c_{ij}$ costul construirii unui arbore binar de căutare optimal care are vârfurile din secvenţa $(a_{i+1}, a_{i+2}, \ldots, a_j)$ şi pseudovârfurile din secvenţa $(e_i, e_{i+1}, \ldots, e_j)$.

MATEMATICA

**Lema 11.3.** *Matricea $C = (c_{i,j})_{i,j=0,\ldots,n}$ poate fi calculată cu ajutorul recurenței*

$$c_{i,j} = \begin{cases} \min_{i+1 \le k \le j}\{c_{i,k-1} + c_{k,j} + w_{i,j}\}, & i \neq j \\ 0, & i = j. \end{cases} \quad \text{unde } w_{i,j} = q_i + \sum_{l=i+1}^{j} (p_l + q_l)$$

11.8.2

*Demonstrație.* Fie $(a_1, a_2, \ldots, a_r, \ldots, a_n)$ secvența de noduri sortată crescător. Presupunem că în primul pas se alege ca rădăcină nodul $a_r$.

Aplicând principiul optimalității, subarborele stâng $L$ este optimal și construit cu secvențele $(a_1, a_2, \ldots, a_{r-1})$ și $(e_0, e_1, \ldots, e_{r-1})$ :

$$\text{cost}(L) = \sum_{i=1}^{r-1} p_i * \text{nivel}(a_i) + \sum_{i=0}^{r-1} q_i * (\text{nivel}(e_i) - 1)$$

Analog, subarborele drept $R$ este optimal și construit cu secvențele $(a_{r+1}, a_{r+2}, \ldots, a_n)$ și $(e_r, e_{r+1}, \ldots, e_n)$ :

$$\text{cost}(R) = \sum_{i=r+1}^{n} p_i * \text{nivel}(_i) + \sum_{i=r}^{n} q_i * (\text{nivel}(e_i) - 1)$$

Valorile `nivel()` sunt considerate în subarborii $L, R$.

Costul asociat arborelui $T$ devine:

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + \sum_{i=1}^{r-1} p_i + \sum_{i=0}^{r-1} q_i + \sum_{i=r+1}^{n} p_i + \sum_{i=r}^{n} q_i.$$

Sumele $\sum_{i=1}^{r-1} p_i + \sum_{i=0}^{r-1} q_i + \sum_{i=r+1}^{n} p_i + \sum_{i=r}^{n} q_i$ reprezintă valorile suplimentare care apar datorită faptului că $T$ introduce un nivel nou.

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + q_0 + \sum_{i=1}^{r-1}(p_i + q_i) + q_r + \sum_{i=r+1}^{n} (p_i + q_i)$$

Dacă se utilizează notația $w_{i,j} = q_i + \sum_{l=i+1}^{j} (p_l + q_l)$, rezultă

$$\text{cost}(T) = p_r + \text{cost}(L) + \text{cost}(R) + w_{0,r-1} + w_{r,n} = \text{cost}(L) + \text{cost}(R) + w_{0,n}$$

În urma aplicării definiției lui $c_{i,j}$ se obține:

$$\text{cost}(T) = c_{0,n},\ \text{cost}(L) = c_{0,r-1} \text{ and } \text{cost}(R) = c_{r,n}$$

$$c_{0,n} = c_{0,r-1} + c_{r,n} + w_{0,n} = \min_{1 \le k \le n} \{c_{0,k-1} + c_{k,n} + w_{0,n}\}$$

Prin generalizare rezultă relaţia (11.8.2)

Recurenţa (11.8.2) este utilizată pentru calcularea matricei de valori optime, care la rândul ei, va fi la baza construirii unui arbore binar de căutare optimal. sfdem

**Teorema 11.2.** *Arborele binar de căutare construit pe baza matricei $C$ este optimal.*

*Demonstraţie.* Relaţia (11.8.2) asigură faptul că $c_{0,n}$ este minimum între toate costurile arborilor binari de căutare construiţi cu secvenţa $(a_1, a_2, \ldots, a_n)$. Astfel, arborele binar de căutare construit pe baza matricei $C$ are cost minim deci este optimal. sfdem

### 11.8.3 Implementare

Matricea $C$ se poate calcula aplicând relaţia de recurenţă (11.8.2) pentru $i - j = 1$, apoi $j - i = 2, \ldots, j - i = n$.

Pentru a pregăti informaţiile necesare construirii efective a arborelui binar de căutare optimal, vor fi calculate şi valorile $r_{i,j}$, unde $r_{i,j}$ semnifică indicele nodului rădăcină al subarborelui optimal format din secvenţa $(a_{i+1}, a_{i+2}, \ldots, a_j)$.

Valorile iniţiale sunt $c_{i,i} = 0, w_{i,i} = q_i,\ 0 \le i \le n$. În plus, se foloseşte relaţia $w_{i,j} = p_j + q_j + w_{i,j-1}$.

Algoritmul de calcul al arborelui optimal este următorul:

```
procedure abc_opt(c,r,w,p,q,n)
 for i=1 to n-1 do
   c_{i,i} ← 0; r_{i,i} ← 0; w_{i,i} ← q_i
   c_{i,i+1} ← q_i+p_{i+1}+q_{i+1}; r_{i,i+1} ← i+1; w_{i,i+1} ← q_i+p_{i+1}+q_{i+1}
 w_{n,n} ←q_n; r_{n,n} ← 0; c_{n,n} ← 0;
 for d=2 to n do
   for i=0 to n-d do
     j ← i+d
     w_{i,j} ←w_{i,j-1}+p_j+q_j
     fie k indicele pentru care se obţine valoarea minimă pentru
         {c_{i,h-1}+c_{h,j}; h∈ I=[i+1,j]} xxx
     c_{i,j} ←w_{i,j}+c_{i,k-1}+c_{k,j}
     r_{i,j} ←k
end
```

Evaluare. Se observă uşor că timpul de execuţie este $O(n^3)$.

## 11.9 Prezentarea formală a paradigmei

### 11.9.1 Modelul matematic

Problemele ale căror soluţii se pot obţine prin programarea dinamică sunt probleme de optim. Un prototip de astfel de problemă este următorul:

Să se determine:

$$\text{optim } R(x_0, \ldots, x_{m-1}) \tag{11.6}$$

după $x_0, \ldots, x_{m-1}$, în condiţiile în care acestea satisfac restricţii de forma:

$$g(x_0, \ldots, x_{m-1}) \; ? \; 0 \tag{11.7}$$

unde $? \in \{<, \leq, =, \geq, >\}$.

Prin *optim* înţelegem *min* sau *max*, iar ecuaţia 11.6 se mai numeşte şi *funcţie obiectiv*. Un alt prototip este furnizat de digrafurile ponderate, unde $R(x_0, \ldots, x_{m-1})$ exprimă suma ponderilor arcelor $x_0, \ldots, x_{m-1}$, iar restricţiile impun ca $x_0, \ldots, x_{m-1}$ să fie drum sau circuit cu anumite proprietăţi.

Paradigma programării dinamice propune găsirea valorii optime prin luarea unui şir de decizii $(d_1, \ldots, d_n)$, numit şi *politică*, unde decizia $d_i$ transformă starea (problemei) $s_{i-1}$ în starea $s_i$, aplicând *principiul de optim* [BD62]:

> Secvenţa de decizii optime (politica optimă) care corespunde stării $s_0$ are proprietatea că după luarea primei decizii, care transformă starea $s_0$ în starea $s_1$, secvenţa de decizii (politica) rămasă este optimă pentru starea $s_1$.

Prin stare a problemei înţelegem, de fapt, o subproblemă. Unei stări $s$ îi asociem o valoare $z$ şi definim $f(z)$, astfel încât, dacă starea $s$ corespunde problemei iniţiale, atunci:

$$f(z) = \text{optim } R(x_0, \ldots, x_{m-1})$$

Principiul de optim conduce la obţinerea unei ecuaţii funcţionale de forma [BD62]:

$$f(z) = \underset{y}{\text{optim}} \; [H(z, y, f(T(z, y)))] \tag{11.8}$$

unde:

- $s$ şi $s'$ sunt două stări cu proprietatea că una se obţine din cealaltă aplicând decizia $d$;
- $z$ este valoarea asociată stării $s$;
- $T(z, y)$ calculează valoarea stării $s'$, iar
- $H$ exprimă algoritmul de calcul al valorii $f(z)$ dat de decizia $d$.

Relaţia de mai sus poate fi interpretată astfel: dintre toate deciziile care se pot lua în starea $s$ (sau care conduc la starea $s$), se alege una care dă valoarea optimă în condiţiile în care politica aplicată în continuare (sau până atunci) este şi ea optimă.

Relaţia 11.8 poate fi dovedită utilizând inducţia şi reducerea la absurd. Deoarece este posibil ca principiul de optim să nu aibă loc pentru anumite formulări, este necesară verificarea sa pentru problema supusă rezolvării. Rezolvarea ecuaţiilor recurente 11.8 conduce la determinarea unui şir de decizii ce în final constituie politica optimă prin care se determină valoarea funcţiei obiectiv.

### 11.9.2 Implementare

Principul de optim implică proprietatea de substructură optimă a soluţiei, care afirmă că *soluţia optimă a problemei include soluţiile optime ale subproblemelor sale*. Această proprietate este utilizată la găsirea soluţiilor corespunzătoare stărilor optime. În general, programele care implementează modelul dat de programarea dinamică au două părţi:

1. În prima parte, se determină valorile optime date de şirul de decizii optime, prin rezolvarea ecuaţiilor 11.8. Presupunem că pentru exemplul reţelei de numere valorile sunt memorate într-un tablou unidimensional `f`.
2. În partea a doua se construiesc soluţiile (valorile $x_i$ care dau optimul) corespunzătoare stărilor optime pe baza valorilor calculate în prima parte, utilizând proprietatea de substructură optimă.

Nu se recomandă scrierea unui program recursiv care să calculeze valorile optime. Dacă în procesul de descompunere problemă $\mapsto$ subproblemă, o anumită subproblemă apare de mai multe ori, ea va fi calculată de câte ori apare. Este mult mai convenabil ca valorile optime corespunzătoare subproblemelor să fie memorate într-un tablou şi apoi combinate pe baza ecuaţiilor 11.8 pentru a obţine valoarea optimă a unei supraprobleme. În acest fel, orice subproblemă este rezolvată o singură dată (aici este una dintre diferenţele dintre programarea dinamică şi strategia *divide-et-impera* unde o aceeaşi subproblemă poate fi calculată de mai multe ori), iar determinarea valorilor optime se face de la subproblemele mai mici la cele mai mari (*bottom-up*). Prin memorarea valorilor optime într-un tablou, regăsirea oricăreia dintre ele se face în timpul $O(1)$ ceea ce conduce la un timp scurt de calcul pentru optimul unei supraprobleme. Toate aceste observaţii sunt strâns legate de metoda de derecursivare din subsecţiunea 3.4. Pentru exemplul cu numerele aşezate în reţea valorile corespunzătoare drumurilor optime vor fi memorate într-un tablou unidimensional şi ordinea de calcul va fi de la baza triunghiului spre vârf, adică în ordinea descrescătoare a indicilor.

Complexitatea algoritmului care calculează valorile optime depinde direct de tipul de recursivitate implicat de recurenţele rezultate prin aplicarea principiului de optim. Dacă rezultă o recursivitate liniară atunci ştim de la tehnicile de derecursivare (subsecţiunea 3) că valorile optime ale subproblemelor pot fi calculate în timp liniar şi memorate într-un tablou unidimensional. În cazul recursiei în cascadă lucrurile sunt mai complicate. Dacă adâncimea arborelui corespunzător apelurilor recursive este $n$ atunci rezultă un număr de $2^n$ de subprobleme de rezolvat. În unele cazuri, o redefinire a noţiunii de stare poate conduce la obţinerea unei recursii liniare obţinându-se astfel o reducere drastică a complexităţii – de la exponenţial la polinomial. De asemenea, o reducere la complexitate polinomială (de cele mai multe ori pătratică) se obţine atunci când se poate aplica metoda de derecursivare cu memorarea rezultatelor pentru subprobleme în tabele (subsecţiunea 3.4). Pentru problemele dificile (de exemplu, cele $\mathbb{NP}$-complete) astfel de posibilităţi nu există. Pentru aceste cazuri se caută un mod de memorare cât mai compactă a valorilor optime pentru subprobleme şi metode cât mai eficiente pentru calculul acestora astfel încât, pentru instanţele de dimensiuni rezonabile, determinarea soluţiei să se facă într-un timp acceptabil.

## 11.10 Exerciţii

**Exerciţiul 11.1.** Se consideră următoarea variantă a determinării LEP minime:
Se consideră date o listă de numere întregi $x^0 = (x_1^0, \ldots, x_n^0)$ şi un alt număr întreg $M^0$. Acestea constituie starea iniţială a problemei. O decizie corespunzătoare stării $\langle (x_1, \ldots, x_k), M \rangle$ constă în extragerea a două numere $x_i$ şi $x_j$ din lista $x = (x_1, \ldots, x_k)$, introducerea în locul lor în listă a sumei $x_i + x_j$ şi de asemenea adăugarea acestei sume la $M$. Se obţine o nouă stare $\langle x', M' \rangle$ unde $x' = x \setminus (x_i, x_j) \cup (x_i + x_j)$ şi $M' = M + x_i + x_j$. Problema constă în determinarea unei secvenţe de decizii care conduce la starea finală $\langle (x_1^0 + \cdots + x_n^0), M \rangle$ cu $M$ minim.

Notăm cu LEPOPT($\langle x, M \rangle$) valoarea sumei din starea finală dată de soluţia optimă.

1. Să se arate că[1]:

$$\text{LEPOPT}(\langle x, M \rangle) = \min\{\text{LEPOPT}(\langle x \setminus (u,v) \cup (u+v), M + (u+v) \rangle) \mid u, v \text{ apar ca elemente distincte în } x\} \tag{11.9}$$

2. Utilizând 11.9 să se calculeze LEPOPT($\langle (5, 8, 3, 11), 0 \rangle$). Valorile stărilor intermediare vor fi memorate într-un tabel. Apoi să se deducă din acest tabel secvenţa de decizii optime.
3. Să se proiecteze un algoritm bazat pe paradigma programării dinamice care determină soluţia optimă.
4. Să se arate că stările date de algoritmul greedy satisfac:

$$\text{LEPOPT}(\langle x, M \rangle) = \text{LEPOPT}(\langle x \setminus (u,v) \cup (u+v), M + (u+v) \rangle) \tag{11.10}$$

unde $u = \min x$ şi $v = \min(x \setminus (u))$.

5. Explicaţi diferenţele şi asemănările dintre algoritmul greedy şi cel dat de programarea dinamică.

**Exerciţiul 11.2.** Următorul algoritm, cunoscut sub numele de algoritmul Bellman-Ford, determină drumurile minime într-un digraf ponderat $D = ((V, A), \ell)$ care pleacă dintr-un vârf $i_0$ dat. Pentru fiecare vârf $i$, $d[i]$ va fi lungimea drumului minim de la $i_0$ la $i$ şi $p[i]$ va fi predecesorul lui $i$ pe drumul minim de la $i_0$ la $i$.

```
procedure BellmanFord(D, i0, d, p)
    for i ← 1 to n do
        p[i] ← 0
        d[i] ← ∞
    d[i0] ← 0
    for k ← 1 to n-1 do
        for each ⟨i, j⟩ ∈ A do
            if (d[j] > d[i]+ℓ[i,j])
```

[1] Aici privim listele ca reprezentări ale *multimulţimilor*. Într-o multimulţime $U$, un element $a$ poate apărea de mai multe ori. Astfel, o multimulţime $U$ peste $S$ poate fi definită ca o funcţie $U : S \to \mathbb{N}$, unde $U(a)$ reprezintă de câte ori apare elementul $a \in S$ în $U$. Operaţiile peste multimulţimi se definesc în mod natural. De exemplu, $a \in U \iff U(a) > 0$, $(U \cup V) : S \to \mathbb{N}$ este dată prin $(U \cup V)(a) = U(a) + V(a)$ etc.

```
                then d[j] ← d[i] + ℓ[i,j]
                     p[j] ← i;
    for each ⟨i,j⟩ ∈ A do
        if (d[j] > d[i]+ℓ[i,j])
           then throw 'exista drum de lungime negativa'
end
```

Se cere:

1. Să se definească noţiunea de stare (subproblemă) pentru problema drumurilor minime cu sursa precizată.
2. Să se aplice principiul de optim pentru a obţine relaţia de recurenţă.
3. Să se precizeze ordinea în care sunt rezolvate subproblemele.
4. Să se arate că, dacă digraful $D$ nu are circuite negative, atunci algoritmul Bellman-Ford determină corect drumurile minime care pleacă din $i_0$.
5. Să se determine complexitatea algoritmului Bellman-Ford.

**Exerciţiul 11.3.** (*Înmulţirea optimă a unui şir de matrici.*) Se consideră un şir $(A_1, \ldots, A_n)$ de matrici, unde $A_i$ este de dimensiune $p_i \times p_{i+1}$. Se doreşte calcularea produsului $A_1 A_2 \cdots A_n$, utilizând un subprogram de înmulţire a două matrici. Datorită asociativităţii, există mai multe posibilităţi de înmulţire a matricelor două câte două. Numărul posibilităţilor este egal cu numărul expresiilor complet parantetizate. De exemplu, pentru $n = 4$ există cinci posibilităţi. Înmulţirea a două matrici de dimensiuni $p \times q$ şi respectiv $q \times r$ necesită $pqr$ înmulţiri. Problema constă în determinarea unei ordini de înmulţire a matricelor două câte două (echivalent, a unei expresii complet parantetizate) care dă numărul minim de înmulţiri.

1. Pentru $1 \leq i \leq j \leq n$, se notează cu $m[i,j]$ numărul minim de înmulţiri necesare pentru a calcula $A_i \cdots A_j$. Să se arate că:

$$m[i,j] = \begin{cases} 0 & \text{, dacă } i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_i p_{k+1} p_{j+1}\} & \text{, dacă } i < j \end{cases}$$

2. Să se scrie un program care determină valorile $m[i,j]$. Care este complexitatea algoritmului descris de program?
3. Să se proiecteze o structură de date pentru reprezentarea unei ordini de înmulţire (unei expresii complet parantetizate).
4. Să se scrie un program care, având la intrare valorile calculate la punctul 2, determină o ordine de înmulţire optimă.
5. Să se modifice algoritmul de la 2 astfel încât să determine simultan valorile $m[i,j]$, dar şi ordinea de înmulţire optimă.

**Exerciţiul 11.4.** Considerăm noţiunea de *subsecvenţă* în sensul definiţiei din 11.6.1

1. Să se arate că problema calculului celei mai lungi subsecvenţe comune a două secvenţe $x = (x_1, \ldots, x_m)$ şi $y = (y_1, \cdots, y_n)$ date este echivalentă cu determinarea distanţei dintre şiruri calculată considerând numai operaţiile de inserare şi ştergere.

2. Să se proiecteze un algoritm pentru determinarea celei mai lungi subsecvenţe comune utilizând paradigma programare dinamică.

**Exerciţiul 11.5.** [CLR93] Fie $P$ un poligon convex în plan. O triangularizare a poligonului $P$ este o mulţime $T$ de diagonale ale lui $P$ care împarte interiorul poligonului în triunghiuri cu interioarele disjuncte (de aici rezultă că diagonalele nu se intersectează). Laturile unui triunghi sunt laturi sau diagonale complete ale poligonului. Peste mulţimea tuturor triunghiurilor ce participă la cel puţin o triangularizare se consideră dată o funcţie de ponderi $w$. Ca exemplu, se poate considera drept pondere a unui triunghi suma lungimilor laturilor sale. Să se scrie un program care să determine o triangularizare pentru care suma ponderilor este minimă.
*Indicaţie.* Presupunem $P = v_1 \dots v_n$ ($v_i$ sunt vârfurile poligonului). Unei triangularizări îi putem asocia un arbore binar definit recursiv astfel: rădăcina arborelui este latura $v_1v_n$. Fie $v_1v_nv_i$ triunghiul din triangularizare care are pe $v_1v_n$ ca latură. Dacă $v_1v_i$ este diagonală, atunci subarborele aflat la stânga va fi cel corespunzător subpoligonului mărginit de $v_1v_i$, aflat în partea opusă triunghiului şi cu rădăcina $v_1v_i$. Dacă $v_1v_i$ este latură a poligonului (*i.e.*, $i = 2$), atunci subarborele aflat la stânga va fi format numai din nodul $v_1v_i$. Analog, dacă $v_nv_i$ este diagonală, atunci subarborele aflat la dreapta va fi cel corespunzător subpoligonului mărginit de $v_nv_i$, aflat în partea opusă triunghiului şi cu rădăcina $v_nv_i$. Dacă $v_nv_i$ este latură a poligonului (*i.e.*, $i = n-1$), atunci subarborele aflat la dreapta va fi format numai din nodul $v_nv_i$. Un exemplu de construire a arborelui este arătat în figura 11.9. În continuare se raţionează la fel ca la înmulţirea optimă a matricelor (exerciţiul 11.3).

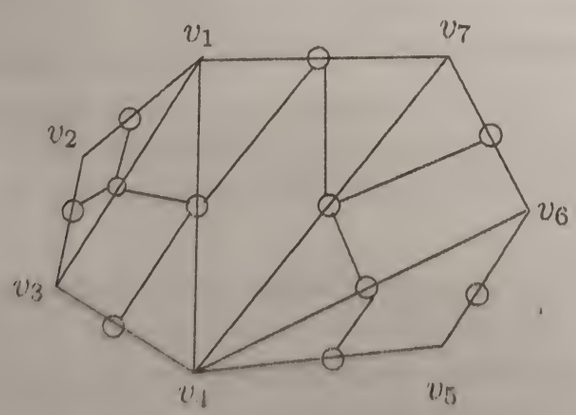



Figura 11.9: Triangularizare şi arborele binar ataşat

**Exerciţiul 11.6.** Fie $p_1, \dots, p_n$ $n$ puncte în plan. Presupunem $p_i = (x_i, y_i)$ cu $x_i \neq x_j$, dacă $i \neq j$. Un *traseu monoton* este un drum care uneşte cel mai din stânga punct cu cel mai din dreapta şi este format din segmente ce unesc punctele din $\{p_1, \dots, p_n\}$ şi pot fi parcurse toate în acelaşi sens (de exemplu, de la stânga la dreapta). Un *tur bitonic* este format din reuniunea a două trasee monotone care conectează toate cele $n$ puncte. Un exemplu de tur bitonic este dat în figura 11.10. Să se scrie un program care determină un tur bitonic de lungime minimă.
*Indicaţie.* Se vor explora punctele de la stânga la dreapta, menţinându-se cele două trasee optime.

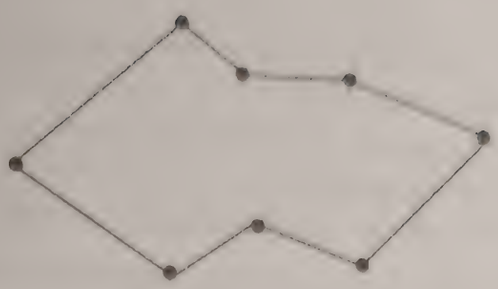


Figura 11.10: Tur bitonic

**Exerciţiul 11.7.** [CLR93] Se consideră problema scrierii la imprimantă într-un mod estetic a unui paragraf. Textul de intrare este o secvenţă de $n$ cuvinte $w_1, \ldots, w_n$ de lungimi $\ell_1, \ldots, \ell_n$, respectiv. Lungimile cuvintelor sunt măsurate în caractere. Se doreşte scrierea acestui paragraf într-un mod cât mai estetic, astfel încât fiecare linie conţine cel mult $M$ caractere. Criteriul prin care se măsoară „estetica" textului este următorul. Dacă o linie conţine cuvintele de la $w_i$ la $w_j$, atunci numărul de spaţii de la sfârşitul liniei este $M - j + i - \sum_{k=i}^{j} \ell_k$. Un text scris estetic minimizează suma, după toate liniile exceptând ultima, a cuburilor numerelor de spaţii extra de la fiecare sfârşit de linie. Să se scrie un program care scrie la imprimantă paragraful dat respectând acest criteriu de esteticitate descris mai sus. Se presupune că $M \leq 80$ şi că un paragraf are cel mult 20 de linii.

**Exerciţiul 11.8.** [CLR93] Când un terminal „inteligent" modifică o linie a unui text, înlocuind şirul „sursă" $x[1..m]$ cu şirul „destinaţie" $y[1..n]$, există câteva operaţii de bază prin care realizează această modificare:

1. un caracter al textului-sursă poate fi şters (`delete` $c$), înlocuit (`replace` $c$ `by` $c'$) sau copiat (`copy` $c$) în textul-destinaţie;
2. un caracter este inserat în textul-destinaţie (`insert` $c$);
3. două caractere vecine din textul-sursă sunt interschimbate în timpul copierii în textul-destinaţie (`twidle` $cc'$ `into` $c'c$;
4. după efectuarea tuturor operaţiilor de tipul celor de mai sus, astfel încât textul-destinaţie este complet, sufixul textului-sursă este şters (`kill` ...).

Considerăm ca exemplu modificarea cuvântului `algorithm` în `altruistic`:

| Operaţie | text sursă | Text destinaţie |
|---|---|---|
| `copy a` | `lgorithm` | `a` |
| `copy l` | `gorithm` | `al` |
| `replace g by t` | `orithm` | `alt` |
| `delete o` | `rithm` | `alt` |
| `copy r` | `ithm` | `altr` |
| `insert u` | `ithm` | `altru` |
| `insert i` | `ithm` | `altrui` |
| `insert s` | `ithm` | `altruis` |
| `twidle it into ti` | `hm` | `altruisti` |
| `insert c` | `hm` | `altruistic` |
| `kill hm` | | `altruistic` |

Fiecare operaţie ≠ `kill` are asociat un cost. Se presupune, de exemplu, că operaţia de înlocuire are costul mai mic decât suma costurilor operaţiilor de ştergere şi de inserare. Costul unei operaţii de modificare este suma costurilor operaţiilor individuale. Să se scrie un program care, având la intrare textele-sursă şi destinaţie, determină secvenţa de operaţii care modifică sursa în destinaţie cu un cost minim.

**Exerciţiul 11.9.** [CLR93] Profesorul McKenzie este consultat de preşedintele companiei A&B Company pentru a planifica o petrecere cu angajaţii companiei. Relaţia şef-subaltern din companie poate fi reprezentată printr-un arbore în care rădăcina este preşedintele companiei. La angajare, în urma unui interviu, fiecărui angajat i s-a atribuit un coeficient de „jovialitate", care este un număr real. Pentru a face petrecerea mai nostimă, s-a decis să nu participe nici o pereche formată dintr-un angajat şi şeful lui direct.

1. Să se scrie un program care-l ajută pe profesor să planifice petrecerea astfel încât coeficientul total de jovialitate (= suma coeficienţilor individuali ai persoanelor care participă la petrecere) să fie maxim.
2. Să se modifice programul de la 1, astfel încât preşedintele companiei să participe la petrecerea companiei sale.

## 11.11 Referinţe bibliografice

În scrierea acestui capitol am fost cel mai mult influenţati de [CLR93, CLR00, HS84]. Mai pot fi consultate [LG86, Sed88, Sah98].

# Capitolul 12

# *Backtracking* şi *branch-and-bound*

*Backtracking* şi *branch-and-bound* sunt două strategii care îmbunătăţesc căutarea exhaustivă. Un algoritm de căutare exhaustivă este definit după următoarea schemă: 1) se defineşte spaţiul *soluţiilor candidat (fezabile)* $\mathbb{U}$; 2) cu ajutorul unui algoritm de enumerare se selectează acele soluţii candidat care sunt soluţii ale problemei. În contrast cu căutarea exhaustivă, paradigmele *backtracking* şi *branch-and-bound* propun o căutare sistematică în spaţiul soluţiilor. Ambele paradigme au la bază următoarele ingrediente:

1. Spaţiul soluţiilor este organizat ca un arbore cu rădăcină astfel încât există o corespondenţă bijectivă între vârfurile de pe frontiera arborelui şi soluţiile candidat. Soluţia candidat este descrisă de drumul de la rădăcină la vârful de pe frontieră corespunzător. Un exemplu de organizare a spaţiului soluţiilor ca arbore este reprezentat în figura 12.1.
2. Căutarea sistematică se realizează cu ajutorul unui algoritm de explorare sistematică a acestui arbore.
3. Vârfurile arborelui sunt clasificate astfel:
   a) un vârf *viabil* este un vârf pentru care sunt şanse să se găsească o soluţie a problemei explorând subarborele cu rădăcina în acel vârf;
   b) un vârf *activ (live)* este un vârf care a fost vizitat cel puţin o dată de algoritmul de explorare şi urmează să mai fie vizitat cel puţin încă o dată;
   c) un vârf activ după ce a fost vizitat ultima dată de algoritmul de explorare devine *inactiv (death)*.
4. Sunt explorate numai subarborii cu rădăcini viabile. În felul acesta se evită procesarea inutilă a subarborilor pentru care suntem siguri că nu conţin soluţii.
5. Cele două paradigme, *backtracking* şi *branch-and-bound*, diferă doar prin modul în care explorează lista vârfurilor viabile din arbore.

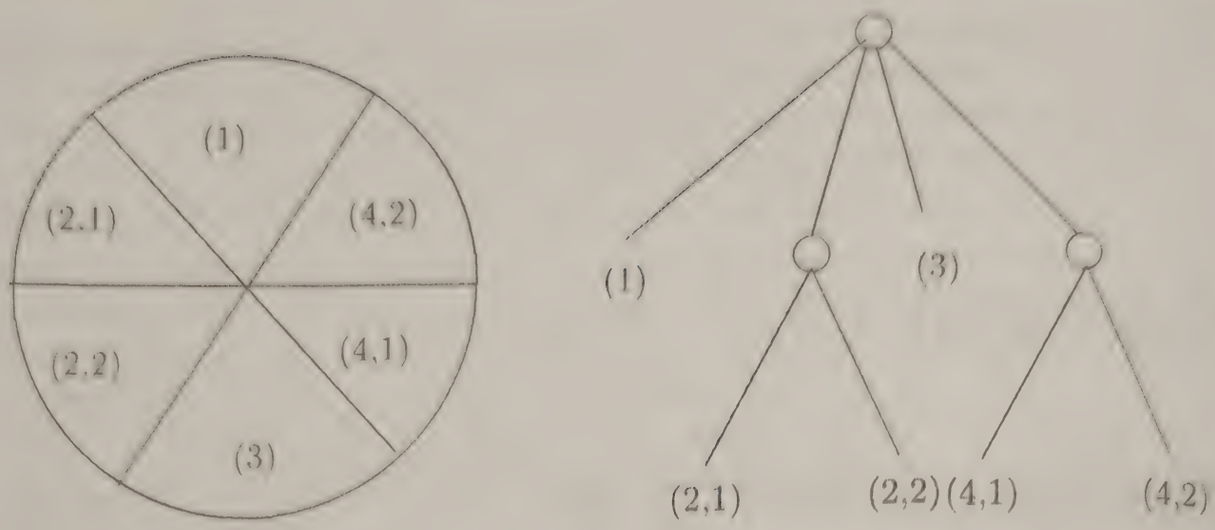



Figura 12.1: Spaţiului soluţiilor ca arbore

## 12.1 Organizarea spaţiului soluţiilor candidat

Metodele *backtracking* şi *branch-and-bound* rezolvă o problemă prin căutarea sistematică a soluţiei în spaţiul soluţiilor candidat. Conceptual, această căutare foloseşte o organizare a spaţiului soluţiilor candidat sub forma unui arbore. Mulţimea elementelor produsului cartezian, submulţimile unei mulţimi, mulţimea permutărilor unei mulţimi şi drumurile într-un graf sunt spaţiile de căutare cele mai utilizate. Din acest motiv vom prezenta în secţiunile care urmează organizarea acestora sub formă de arbore.

### 12.1.1 Produsul cartezian

Considerăm mai întâi cazul în care toate elementele produsului cartezian au aceeaşi lungime. Fără să restrângem generalitatea, putem considera problema enumerării elementelor produsului cartezian $\{0, \ldots, m-1\}^n = \{0, \ldots, m-1\} \times \cdots \times \{0, \ldots, m-1\}$ (de $n$ ori). Dimensiunea spaţiului soluţiilor candidat este $m^n$. Mulţimea $\{0, \ldots, m-1\}^n$ poate fi reprezentată printr-un arbore cu $n$ nivele în care fiecare vârf intern are exact $m$ succesori, iar vârfurile de pe frontieră corespund elementelor mulţimii. De exemplu, arborele corespunzător cazului $m = 2$ şi $n = 3$ este reprezentat grafic în figura 12.2.

Fiecare vârf în arbore este identificat de drumul de la rădăcină la el: notăm acest drum cu $(x_0, \ldots, x_k)$. Pentru vârfurile de pe frontieră avem $k = n - 1$, iar drumul $(x_0, \ldots, x_{n-1})$ este un element al produsului cartezian. Algoritmul pe care îl prezentăm va simula generarea acestui arbore prin parcurgerea DFS. Deoarece lista vârfurilor succesoare pot fi determinate printr-un calcul foarte simplu, stiva este reprezentată de o variabilă simplă $k$, despre care putem gândi că indică poziţia vârfului stivei.

```
procedure enumProdCart(n, m)
    k ← 0
    x[0] ← -1
```

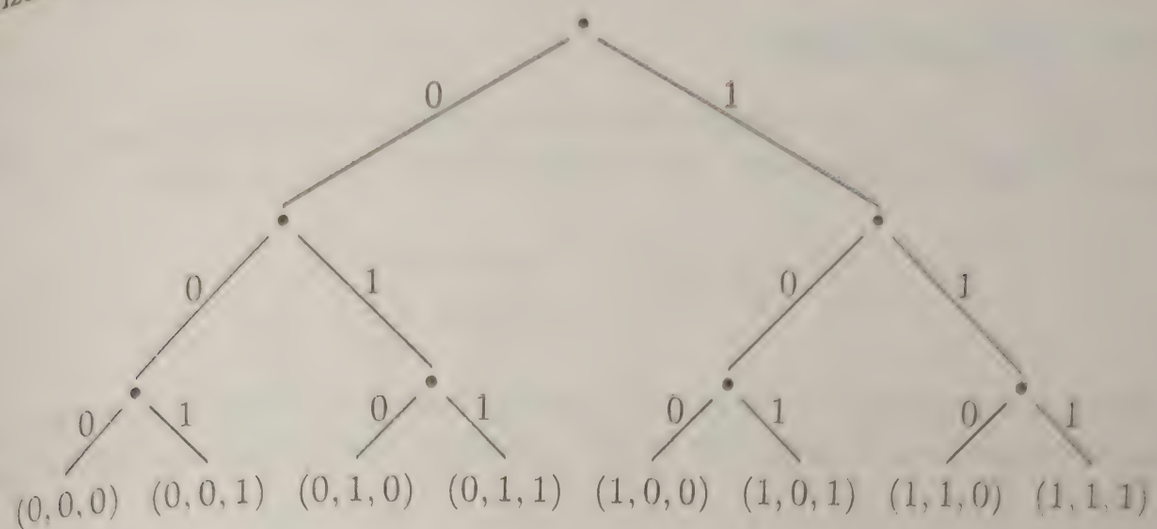



Figura 12.2: Arborele produsului cartezian

```
   while (k ≥ 0) do
      if (x[k] < m-1)
         then x[k] ← x[k]+1
              if (k = n)
                 then scrieElement(x, n)
                 else k ← k+1
                      x[k] ← -1
         else k ← k-1
end
```

Varianta recursivă a programului `enumProdCart` este următoarea:

```
procedure enumProdCartRec(x, k)
   for j ← 0 to m-1 do
       x[k] ← j
       if (k = n-1)
          then scrieElement(x, n)
          else enumProdCartRec(x, k+1)
end
```

Enumerarea tuturor elementelor produsului cartezian cu varianta recursivă se face executând apelul:

```
enumProdCartRec(x, 0)
```

Considerăm acum cazul când mulţimile care definesc produsul cartezian nu au acelaşi cardinal. Presupunem din nou, fără să restrângem generalitatea, că se doreşte enumerarea elementelor produsului cartezian $\{0,\ldots,m_1-1\}\times\cdots\times\{0,\ldots,m_n-1\}$. Programul de enumerare se obţine prin înlocuirea în programul corespunzător de mai sus a variabilei simple `m` cu componenta tablou `m[k]`, unde `m[k]` $= m_k$.

### 12.1.2 Submulţimile

Fie $A = \{a_0, \ldots, a_{n-1}\}$ o mulţime finită cu $n$ elemente. O submulţime $S \subseteq A$ poate fi reprezentată de vectorul său caracteristic $(x[i] \mid 0 \leq i < n)$, unde

$$x[i] = \begin{cases} 1 & \text{, dacă } a_i \in S, \\ 0 & \text{, dacă } a_i \notin S. \end{cases}$$

Acum, enumerarea submulţimilor mulţimii $A$ este echivalentă cu enumerarea elementelor produsului cartezian $\{0,1\}^n$. Dimensiunea spaţiului soluţiilor candidat este $2^n$.

### 12.1.3 Permutările

Notăm cu $S_n$ mulţimea permutărilor mulţimii $\{0, 1, \ldots, n-1\}$. Dimensiunea spaţiului soluţiilor candidat este $n! = 1 \cdot 2 \cdot \cdots \cdot n$.

#### 12.1.3.1 Varianta I

Spaţiul soluţiilor este reprezentat ca un arbore în care orice drum de la rădăcină la frontieră descrie o permutare. Arborele pentru $S_3$ este reprezentat în figura 12.3. Deoarece orice permutare din $S_n$ este un element al produsului cartezian $\{0, 1, \ldots, n-1\}^n$ ce satisface anumite restricţii, rezultă că putem utiliza algoritmul *backtracking* descris în secţiunea 12.1.1. Un vârf $x[k]$ este *viabil*, dacă satisface următoarea proprietate:

$$0 \leq x[k] \leq n-1 \wedge (\forall i) 0 \leq i < k \Rightarrow x[i] \neq x[k]$$

Verificarea condiţiei de mai sus este destul de costisitoare, $O(k)$. Deoarece această condiţie este verificată pentru toate valorile candidat pentru $x[k]$, rezultă că timpul cel mai nefavorabil pentru determinarea unui $x[k]$ viabil este $O(k^2)$. Performanţele algoritmului pot fi îmbunătăţite dacă se consideră un tablou suplimentar $(y[j] \mid 0 \leq j < n)$ cu semnificaţia:

$$y[j] = \begin{cases} 1, & \text{dacă există } i \text{ cu proprietatea } x[i] = j, \\ 0, & \text{altfel.} \end{cases}$$

Căutarea următorului $x[k]$ viabil se face incrementând $x[k]$ până când are loc condiţia $y[x[k]] = 0$, după care $y[x[k]]$ devine 1. Decrementarea lui $k$ este, de asemenea, precedată de atribuirea valorii 0 lui $y[x[k]]$.

#### 12.1.3.2 Varianta II

Se bazează pe o altă reprezentare arborescentă a permutărilor. Cazurile practice în care această enumerare este utilizată ca bază pentru strategia *backtracking* sunt mai rare, dar enumerarea simplă a permutărilor este mai eficientă.

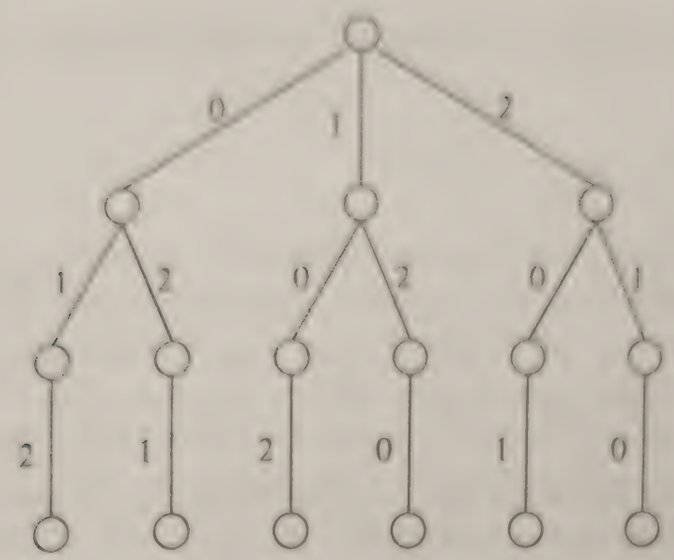



Figura 12.3: Permutări – varianta I ($n = 3$)

**Enumerarea recursivă.** Scrierea unui program recursiv pentru generarea permutărilor trebuie să aibă ca punct de plecare o definiţie recursivă pentru $S_n$. Dacă $(i_0, \ldots, i_{n-1})$ este o permutare din $S_n$ cu $i_k = n-1$ atunci $(\ldots i_{k-1}, i_{k+1}, \ldots)$ este o permutare din $S_{n-1}$. Deci orice permutare din $S_n$ se obţine dintr-o permutare din $S_{n-1}$ prin inserţia lui $n-1$ în una din cele $n$ poziţii posibile. Evident, permutări distincte din $S_{n-1}$ vor produce permutări diferite în $S_n$. Aceste observaţii conduc la următoarea definiţie recursivă:

$$S_1 = \{(0)\}$$
$$S_n = \{(i_0, \ldots, i_{n-2}, n-1), (i_0, \ldots, n-1, i_{n-2}), \ldots, (n-1, i_0, \ldots, i_{n-2}) \mid (i_0, \ldots, i_{n-2}) \in S_{n-1}\}$$

Generalizăm prin considerarea mulţimii $S_n(\pi, k)$ a permutărilor din $S_n$ ce pot fi obţinute din permutarea $\pi \in S_k$. Pentru $S_n(\pi, k)$ avem următoarea definiţie recursivă:

$$S_n(\pi, k) = S_n((i_0, \ldots, i_{k-1}, k), k) \cup \cdots \cup S_n((k, i_0, \ldots, i_{k-1}), k)$$

unde $\pi = (i_0, \ldots, i_{k-1})$. Are loc $S_n = S_n((0), 0)$ şi $S_n(\pi, n-1) = \{\pi\}$. Vom scrie un subprogram recursiv pentru calculul mulţimii $S_n(\pi, k)$ şi apoi vom apela acest subprogram pentru determinarea lui $S_n$. Pentru reprezentarea permutărilor utilizăm tablouri unidimensionale. Subprogramul recursiv care calculează mulţimea $S_n(\pi, k)$ are următoarea descriere:

```
procedure enumPermRec(p, k)
   if (k = n-1)
      then scriePerm(p,n)
      else p[k] ← k
           for i ← k-1 downto 0 do
               enumPermRec(p,k+1)
               swap(p[i+1],p[i])
           enumPermRec(p,k+1)
end
```

Enumerarea tuturor celor $n!$ permutări se realizează prin execuția următoarelor două instrucțiuni

```
p[0] ← 0
enumPermRec(p, 0)
```

**Enumerarea nerecursivă.** Metodei recursive i se poate ataşa un arbore ca în figura 12.4. Fiecare vârf intern din arbore corespunde unui apel recursiv. Vârfurile de pe frontieră corespund permutărilor din $S_n$. Ordinea apelurilor recursive coincide cu ordinea dată de parcurgerea DFS a acestui arbore. Dacă vom compara programul recursiv care generează permutările cu varianta recursivă a algoritmului DFS, vom observa că ele au structuri asemănătoare. Este normal să fie aşa pentru că programul de enumerare a permutărilor realizează acelaşi lucru: parcurgerea mai întâi în adâncime a arborelui din figura 12.4. Deci şi varianta nerecursivă a algoritmului de generare a permutărilor poate fi obţinut din varianta nerecursivă a algoritmului DFS. Locul tabloului $p$ din DFS este luat de o funcţie $f(k, i, p)$ care pentru o permutare $(p[0], \ldots, p[k-1])$ (aflată pe nivelul $k-1$ în arbore) determină al $i$-lea succesor, $0 \leq i \leq k$. Deoarece pentru orice permutare, corespunzătoare unui vârf de pe nivelul $k \geq 1$ în arbore, putem determina permutarea din vârful părinte, rezultă că nu este necesară memorarea permutărilor în stivă. Astfel, stiva va memora, pentru fiecare nivel din arbore, indicele succesorului ce urmează a fi vizitat.

```
procedure enumPerm(n)
   k ← 0
   S[0] ← 0
   while (k ≥ 0) do
      if (S[k] ≥ 0)
         then f(k, S[k], p)
              S[k-1] ← S[k-1]-1
              if (k = n-1)
                 then scriePerm(p,n)
                 else k ← k+1
                      S[k] ← k
         else aux ← p[0]
              for i ← 0 to k-1 do
                  p[i] ← p[i+1]
              p[k] ← aux
              k ← k-1
end
```

Funcţia $f(k, i, p)$ este calculată de următorul subprogram:

```
function f(k, i, p)
   if (i = k)
      then p[k] ← k
      else aux ← p[i+1]
           p[i+1] ← p[i]
           p[i] ← aux
end
```

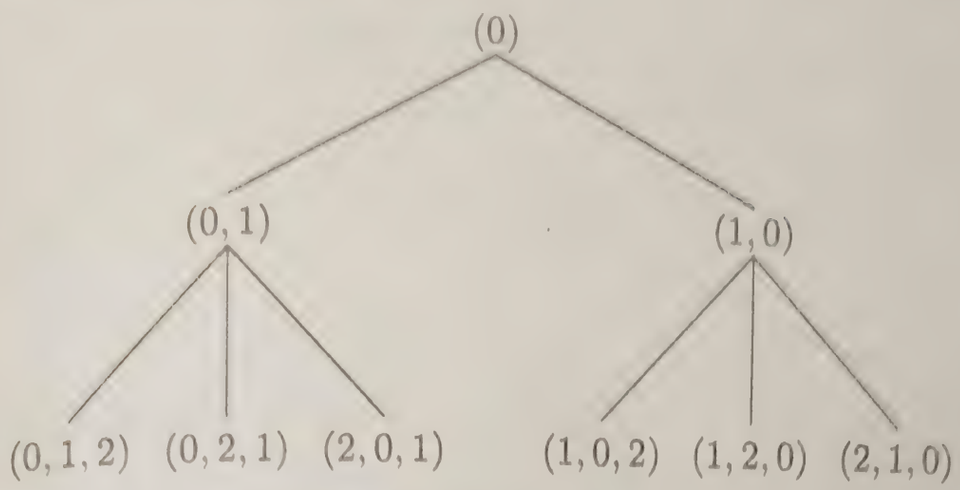



Figura 12.4: Arborele permutărilor generat de metoda recursivă

**Exerciţiul 12.1.1.** Să se scrie un subprogram care, pentru numerele naturale $i$ şi $n$ date, determină a $i$-a permutare (în ordinea lexicografică) din $S_n$.

**Observaţie.** Algoritmul de mai sus poate fi îmbunătăţit din punctul de vedere al timpului de execuţie. Mai întâi să notăm faptul că orice algoritm de enumerare a permutărilor are timpul de execuţie $O(n!) = O(n^n)$. Ideea este de a găsi un algoritm care să efectueze cât mai puţine operaţii pentru determinarea permutării succesoare. Execuţia a $c' \cdot n!$ operaţii în loc de $c \cdot n!$, $c' < c$, semnifică, de fapt, o reducere cu $(c - c') \cdot n!$ a timpului de execuţie. Un astfel de algoritm este obţinut după cum urmează. În arborele din figura 12.4 se schimbă ordinea succesorilor permutării (1,0). Ordinea permutărilor de pe orice nivel din noul arbore are proprietatea că oricare două permutări succesive diferă printr-o transpoziţie de poziţii vecine. Dacă se reuşeşte generarea permutărilor direct în această ordine, fără a simula parcurgerea arborelui, atunci se obţine un program care generează permutările cu număr minim de operaţii. Regula generală prin care se obţine această ordine este următoarea (figura 12.5):

> La fiecare nivel din arborele apelurilor recursive, succesorii vârfurilor de rang par îşi schimbă ordinea, astfel încât cel mai din stânga devine cel mai din dreapta şi cel mai din dreapta devine cel mai din stânga.

Evitarea simulării parcurgerii arborelui se realizează prin utilizarea unui vector de „direcţii", $d = (d[k] \mid 0 \leq k < n)$, cu următoarea semnificaţie:

- $d[k] = +1$ dacă permutările succesoare permutării $(p[0], \ldots, p[k-1])$ sunt generate în ordinea $(p[0], \ldots, p[k-1], k), \ldots, (k, p[0], \ldots, p[k-1])$;
- $d[k] = -1$ dacă permutările succesoare permutării $(p[0], \ldots, p[k-1])$ sunt generate în ordinea $(k, p[0], \ldots, p[k-1]), \ldots, (p[0], \ldots, p[k-1], k)$;

În acest mod, vectorul $d$ descrie complet drumul de la rădăcină la un grup de permutări pentru care transpoziţia se aplică în aceeaşi direcţie. Determinarea indicelui $i$ la care se aplică transpoziţia se poate face utilizând un tablou care memorează permutarea inversă. Dacă notăm acest tablou cu *pinv*, atunci, utilizând relaţia $p[pinv[k]] = k$, obţinem că locul $i$ unde se află $k$ este $pinv[k]$. Noua poziţie a lui $k$ va fi $i + d[k]$.

sfobs

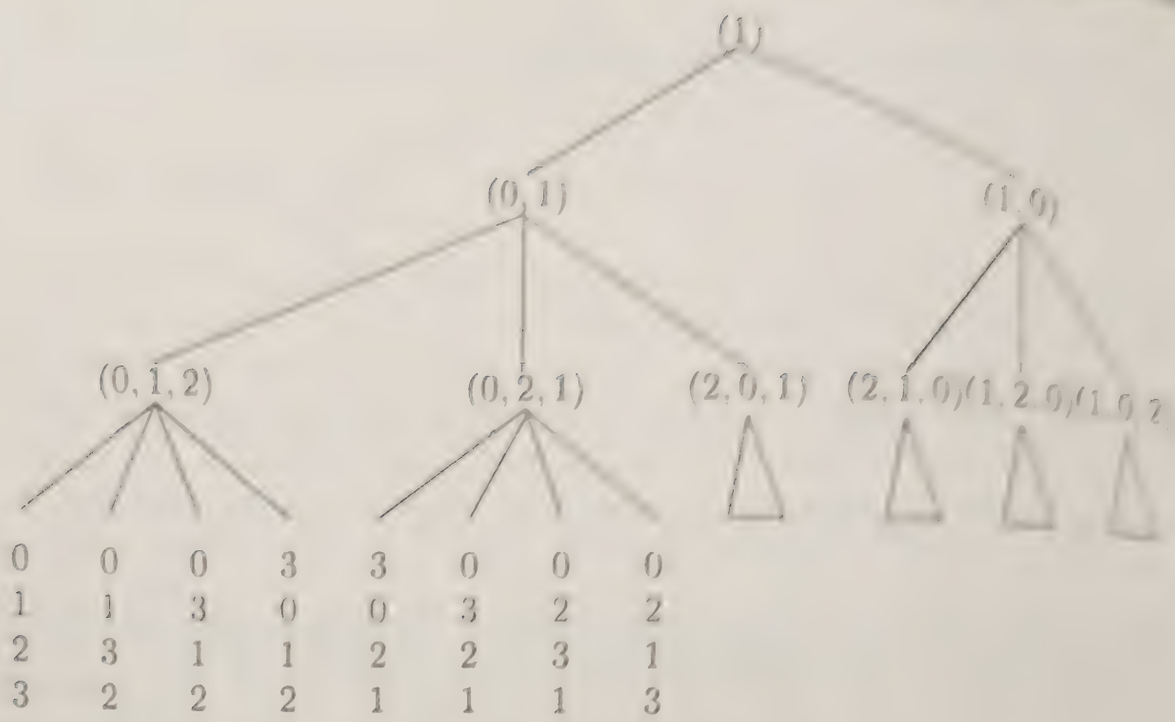



Figura 12.5: Arborele permutărilor modificat

### 12.1.4 Drumurile într-un (di)graf

În multe probleme practice, spaţiul soluţiilor candidat este dat de drumurile într-un (di)graf. Arborele reprezentând drumurile care pleacă dintr-un vârf fixat $i_0$ poate fi obţinut prin inserarea liniei m: în algoritmul DFS

- varianta nerecursivă:

```
procedure drumuriDigraf(D, i0, viziteaza())
   for ← 0 to D.n-1 do
       p[i] ← D.a[i].prim
   SB ← stivaVida()
   push(SB, i0)
   viziteaza(i0)
   while (not esteVida(SB)) do
      i ← top(SB)
      if (p[i] = NULL)
         then pop(SB)
              /* se reia procesul de explorare a vecinilor */
m:            p[i] ← D.a[i].prim
         else j ← p[i]->elt
              p[i] ← p[i]->succ
              if j ∉ SB
                 then viziteaza(j)
                      push(SB, j)
end
```

- varianta recursivă:

```
procedure drumuriDigrafRec(i);
   viziteaza(i)
```

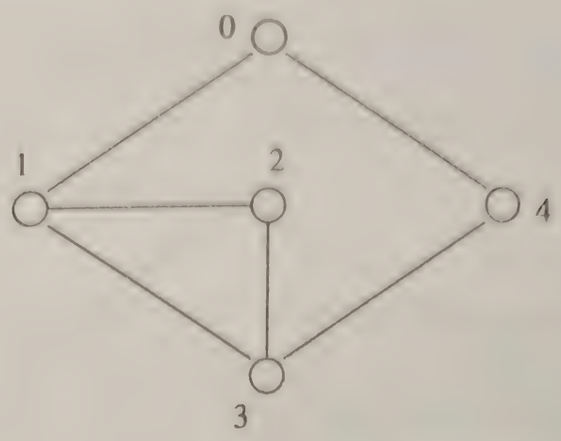



Figura 12.6: Graf

```
    push(SB, i)
    S[i] ← 1
    for fiecare vârf j adiacent cu i do
        if (j ∉ SB) then drumuriDigrafRec(j)
    pop(SB)
end
```

Dacă presupunem că graful din figura 12.6 este reprezentat ca digraf de listele de adiacenţă:

$$0 \rightarrow (1,4)$$
$$1 \rightarrow (0,2,3)$$
$$2 \rightarrow (1,3)$$
$$3 \rightarrow (1,2,4)$$
$$4 \rightarrow (0,3)$$

atunci arborele care reprezintă toate drumurile ce plecaă din 0 este cel reprezentat în figura 12.7.

Un caz particular îl constituie labirinturile. De exemplu, labirintul din figura 12.8a poate fi reprezentat prin matricea:

$$a = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

care, la rândul ei, este reprezentată de graful din figura 12.8b.

Parcurgerea tuturor drumurilor din labirint care pleacă din $(0,0)$ nu presupune construcţia efectivă a grafului, ci utilizarea directă a matricei $a$. Convenim ca în timpul parcurgerii să punem $a[i,j] = 2$ dacă şi numai dacă $(i,j)$ este în stivă (elementele egale cu 2 din matrice vor descrie stiva $SB$). Presupunem că listele

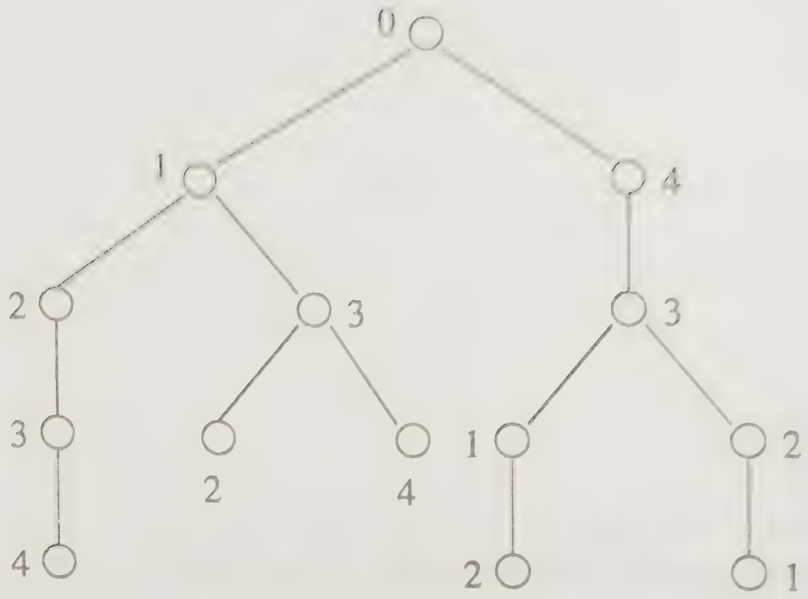



Figura 12.7: Arborele drumurilor din graful 12.6

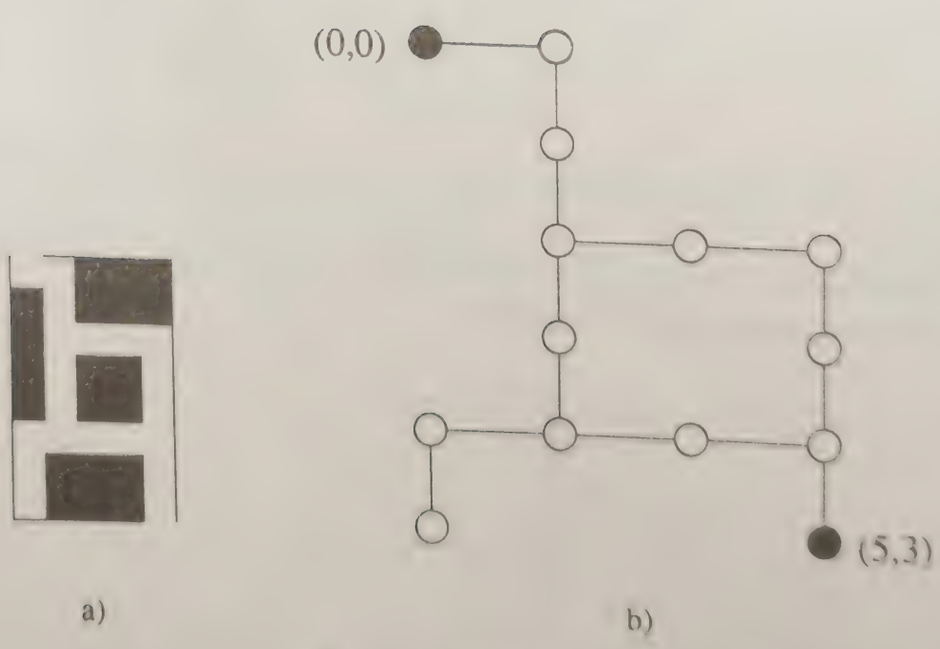



Figura 12.8: Labirint

de adiacenţă sunt reprezentate în ordinea parcurgerii vecinilor în sensul arcelor de ceasornic începând cu vecinul din dreapta. Cel mai simplu este să modificăm varianta recursivă, înlocuind bucla `for each` cu apelurile corespunzătoare vecinilor:

```
procedure drumuriLabirint(i,j);
   viziteaza(i,j)
   a[i,j] ← 2
   if (j < n-1 and a[i,j+1] = 0) then drumuriLabirint(i, j+1)
   if (i < m-1 and a[i+1,j] = 0) then drumuriLabirint(i+1, j)
   if (j > 0 and a[i,j-1] = 0) then drumuriLabirint(i, j-1)
   if (i > 0 and a[i-1,j] = 0) then drumuriLabirint(i-1, j)
   a[i,j] ← 0
end
```

## 12.2 *Backtracking*

Presupunem că o soluţie candidat este soluţie a problemei dacă satisface o anumită condiţie, notată cu *ST* (eSTe), ce poate fi testată în timp polinomial. Putem privi *ST* ca o funcţie $ST : \mathbb{U} \rightarrow Boolean$. Paradigma *backtracking* se bazează pe următoarele reguli:

1. se defineşte o funcţie criteriu prin care se stabileşte dacă un vârf este viabil sau nu;
2. arborele este explorat prin algoritmul DFS;
3. fie $x^k = (x_0, ..., x_k)$ secvenţa care descrie drumul de la rădăcina la vârful curent;
4. dacă vârful curent este pe frontieră, atunci se verifică dacă $x^k$ este soluţie;
5. dacă vârful curent nu este pe frontieră, se alege următorul succesor viabil (dacă există).

Dacă $x^k = (x_0, ..., x_k)$ este secvenţa care descrie drumul de la rădăcina la vârful curent, se notează cu $T(x^k)$ mulţimea tuturor valorilor posibile pentru $x_{k+1}$, astfel încât secvenţa $x^{k+1} = (x_0, x_1, ..., x_k, x_{k+1})$ descrie de asemenea un drum de la rădăcină către o frunză.

Modelul general de algoritm *bactracking* în varianta nerecursivă are descrierea următoare:

```
procedure backtracking(n)
   k ← 0
   while (k ≥ 0) do
      if (∃ y∈ T(x^k) neîncercat and viabil(y))
         then x_{k+1} ← y
              if ST(x^{k+1})
                 then scrie(x^{k+1})
                 else k ← k+1
         else k ← k-1
end
```

Varianta recursivă poate fi descrisă astfel:

```
procedure backtrackingRec(x^k)
   k ← 0
   for each y∈ T(x^k) neîncercat and viabil(y) do
       x_{k+1} ← y
       if ST(x^{k+1})
          then scrie(x^{k+1})
          else backtrackingRec(x^{k+1})
end
```

Dacă spaţiul soluţiilor candidat este produsul cartezian $\{0, 1, \ldots, m-1\}^n$, atunci algoritmii backtracking au următoarea descriere generală:

1. varianta nerecursivă:

```
procedure backtrack(n, m)
   k ← 0
   x[0] ← -1
   while (k ≥ 0) do
      if (x[k] < m-1)
         then repeat
                 x[k] ← x[k]+1
              until(viabil(x, k) or (x[k]=m-1))
              if (viabil(x, k))
                 then if ((k = n-1) and ST(x))
                         then scrieElement(x,n)
                         else k ← k+1
                              x[k] ← -1
         else k ← k-1
end
```

2. varianta recursivă:

```
procedure backtrackRec(x, k)
   for j ← 0 to m-1 do
       x[k] ← j
       if ((k = n-1) and ST(x))
          then scrieElement(x, n)
          else if (viabil(x, k))
                  then backtrackRec(x, k+1)
end
```

## 12.3 *Branch-and-bound*

Presupunem că spaţiul soluţiilor candidat (fezabile) este $S$ şi, ca şi în cazul paradigmei *backtracking*, este organizat ca un arbore. Rădăcina arborelui corespunde problemei iniţiale (căutării în $S$), iar fiecare vârf intern corespunde unei subprobleme a problemei iniţiale. *Branch-and-bound* dezvoltă acest arbore conform următoarelor reguli:

1. Fiecare vârf viabil este explorat o singură dată.

2. Când un vârf viabil este explorat, toţi fii viabili sunt generaţi şi memoraţi într-o *structură de date de aşteptare* pe care o notăm cu $A$. Iniţial, $A$ include doar rădăcina.
3. Următorul vârf explorat este ales din structura de aşteptare $A$.

Există trei moduri clasice de implementare pentru structura de aşteptare:

- *Coadă.* Arborele este explorat la fel ca în cazul algoritmului BFS.
- *Heap.* Se aplică pentru problemele de optim: pentru probleme de minimizare se alege min-*heap*, iar pentru probleme de maximizare se alege max-*heap*. Se asociază o funcţie *predictor* (valoare de cost aproximativă) pentru fiecare vârf şi valoarea acesteia va constitui cheia în *heap*.
- *Stivă.* Arborele este explorat în lăţime, dar într-o manieră diferită de BFS. De exemplu, dacă se consideră arborele spaţiului de soluţii din figura 12.9, atunci în cazul cozii ordinea explorării vârfurilor este (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11), iar în cazul stivei ordinea este (1, 4, 10, 3, 9, 8, 7, 2, 6, 11, 5). Deşi pare paradoxal, principiul de bază al metodei *branch-and-bound* este respectat şi în cazul stivei: când un vârf viabil este explorat, toţi fiii viabili sunt generaţi şi memoraţi. Astfel, în momentul în care este explorat vârful 1, vârfurile 2, 3 şi 4 sunt generate şi memorate în stivă. Următorul vârf explorat este 4, situat în vârful stivei. Procesul continuă cu acest vârf.

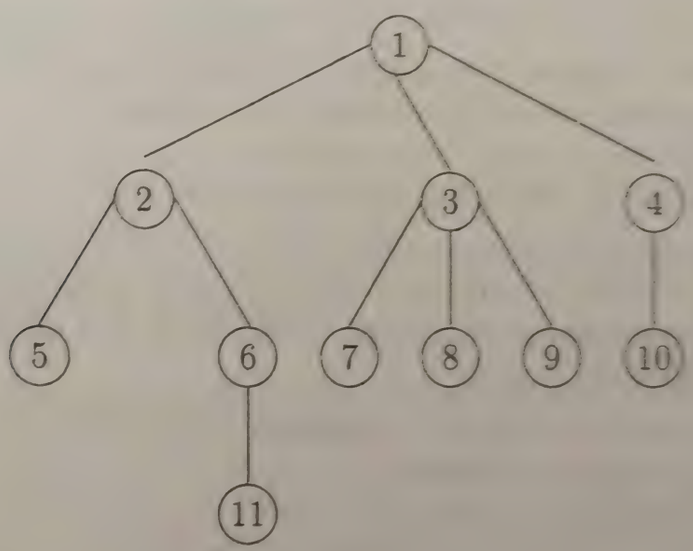



Figura 12.9: Arbore de soluţii pentru *branch-and-bound*

*Branch-and-bound* este termenul generic pentru toate tehnicile care au în comun faptul că vârful curent explorat este complet expandat (se generează toţi fiii şi se memorează în structura de aşteptare), după care se alege pentru explorare, după o strategie oarecare, un nou vârf. Procedura de mai sus se descrie algoritmic astfel:

```
procedure BranchAndBound()
  fie r rădăcina
  A ← {r}
  while A ≠ ∅ do
    selectează v din A
```

```
    A ← A \ {v}
    if v este pe frontieră
      then return solutie(v)
      else fie v_1,...,v_k fiii viabili ai lui v
           A ← A ∪ {v_i | 1 ≤ i ≤ k}
  throw 'Nu există soluţie.'
end
```

### 12.3.1 *Branch-and-bound* pentru probleme de optim

Se aplică problemelor de optim pentru care nu există algoritmi *greedy* sau bazaţi pe programare dinamică. În general, sunt probleme $\mathbb{NP}$-dificile (a se vedea secţiunea 13.3). Se poate aplica şi problemelor care nu sunt de optim, dar care pot fi formulate ca probleme de optim prin adăugarea unei funcţii de cost care să selecteze soluţia (soluţiile) dorite; un astfel de exemplu este dat în secţiunea 12.8. Reamintim că fiecare vârf $v$ descrie o *soluţie potenţială* $x^k$ (drumul de la răcină la $v$); prin $c(v)$ notăm valoarea $c(x^k)$. Mulţimea soluţiilor potenţiale o include pe cea a soluţiilor candidat; presupunem că funcţia $c$ este definită peste soluţiile potenţiale.

### 12.3.2 *Branch-and-bound* cu cost minim

*Branch-and-bound* cu cost minim poate fi descrisă pe scurt astfel:

1. Se consideră o *funcţie de predicţie* $c^*$ definită pentru fiecare vârf $v$ din arbore, care trebuie să satisfacă următoarele proprietăţi:
   a) $c^*(v) \leq c(v)$ pentru fiecare vârf $v$;
   b) dacă $v$ este pe frontieră, atunci $c^*(v) = c(v)$;
   c) dacă $w$ este fiu al lui $v$, atunci $c^*(v) \leq c^*(w)$; prin tranzitivitate, obţinem $c^*(v) \leq c^*(w)$ pentru orice descendent $w$ al lui $v$.
2. Structură de aşteptare $A$ este un min-heap pentru ca la fiecare pas să fie ales elementul cu $c^*(v)$ minim.

Procedura de mai sus se descrie algoritmic astfel, unde operaţiile peste $A$ trebuie citite ca operaţii peste un min-heap:

```
procedure BBCostMin()
  fie r rădăcina
  calculează c*(r)
  A ← {(r, c*(r))}
  while (A ≠ ∅) do
    selectează v din A cu c*(v) minim
    A ← A \ {(v,c*(v)}
    if v este pe frontieră
      then return solutie(v)
      else fie v_1,...,v_k fiii viabili ai lui v
           calculează c*(v_1),...,c*(v_k)
           A ← A ∪ {(v_i,c*(v_i)) | 1 ≤ i ≤ k}
  throw 'Nu există soluţie.'
end
```

Structura standard a funcției $c^*$ este

$$c^*(v) = f(h(v)) + g^*(v)$$

unde:

- $h(v)$ este costul drumului de la rădăcină la $v$ (de exemplu, lungimea drumului);
- $f$ este o funcție crescătoare;
- $g^*(v)$ este costul subestimat al obținerii unei soluții pornind din $v$; $g^*(v)$ va constitui componenta optimistă a costului $c^*(v)$.

Calculul lui $g^*(v)$ nu este întotdeauna simplu. Nu se poate da o formulă generală pentru această funcție.

**Teorema 12.1.** *Dacă $v$ este vârful calculat de* `BBCostMin(r,c*)`*, atunci drumul de la $r$ la $v$ reprezintă soluția de cost minim, adică pentru orice alt vârf $s$ de pe frontieră, avem $c(s) \geq c(v)$.*

*Demonstrație.* Din proprietățile lui $c^*$ rezultă că pentru orice vârf $s$ de pe frontieră $c^*(s) = c(s)$ și pentru orice descendent $w$ al lui $v$, $c^*(v) \leq c^*(w)$. Vârful $v$ este primul vârf de pe frontieră extras din structura de așteptare $A$, deci celelalte vârfuri de pe frontieră se află în structura de așteptare $A$ sau sunt descendenți ai unor vârfuri din această structură. Rezultă:

- $c(v) = c^*(v) \leq c^*(w)$, pentru orice vârf $w$ din structura de așteptare,
- $c^*(w) \leq c^*(s) = c(s)$, pentru orice vârf $s$ de pe frontieră, descendent al unui vârf $w$ din structura de așteptare.

În consecință, pentru orice vârf $s$ de pe frontieră, $c(s) \geq c(v)$. sfdem

### 12.3.3 *Branch-and-bound* cu cost minim și mărginire

Strategia *branch-and-bound* cu cost minim și mărginire este similară cu *branch-and-bound* cu cost minim și poate fi descrisă astfel:

1. Se consideră *funcția de predicție* $c^*$ utilizată de strategia cost minim și o funcție de predicție suplimentară $u$, cu următoarele proprietăți:
   a) $c(v) \leq u(v)$ pentru fiecare vârf $v$;
   b) dacă $v$ este pe frontieră, atunci $c(v) = u(v)$;
   c) dacă $w$ este fiu al lui $v$, atunci $u(w) \leq u(v)$; prin tranzitivitate, obținem $u(w) \leq u(v)$ pentru orice descendent $w$ al lui $v$.
2. Structură de așteptare $A$ este de asemenea un min-*heap* pentru ca la fiecare pas să fie ales tot elementul cu $c^*(v)$ minim.

$u(v)$ este o funcție compusă $f(h(v)) + k^*(v)$, unde $f$ și $h$ sunt definite la fel ca în cazul funcției $c^*$, iar $k^*(v)$ exprimă o estimare supraevaluată (pesimistă) a costului obținerii unei soluții pornind din $v$.

Se pune problema determinării unui vârf $s$ de pe frontieră, pentru care $u(s) = \min\{u(v)\}$.

**Observație.** Pentru un vârf de pe frontieră, $c^*(s) = c(s) = u(s)$. Deci, problema determinării lui $s$ este echivalentă cu problema determinării unui vârf de pe frontieră de cost minim. sfobs

Algoritmul *branch-and-bound* cu cost minim și mărginire este următorul:

```
procedure BBCostMin&Marg()
  fie r rădăcina
  calculează c*(r)
  A ← {(r, c*(r)}
  s ← ∅
  while A ≠ ∅ do
    selectează v din A cu c*(v) minim
    A ← A \ {(v,c*(v)}
    if (c*(v) ≥ u_min)
       then break
       else for each w fiu al lui v
                calculează c*(w)
                if (c*(w) < u_min)
                   A ← A ∪ {(w,c*(w)}
                   if (w este pe frontieră)
                     then u_min ← c*(w)
                          s ← w
                     else calculează u(w)
                          u_min ← min{ u_min, u(w)+ε }
  if s ≠ ∅
    then return solutie(s)
    else throw 'Nu există soluţie.'
end
```

**Lema 12.1.** *Vârfurile $w$, care schimbă valoarea lui $u_{min}$ şi nu sunt pe frontieră, nu pot fi ultimele ce realizează această modificare.*

*Demonstraţie.* Din relaţiile $c^*(w) \leq u(w) < u(w) + \varepsilon = u_{min}$ rezultă faptul că $w$ este un posibil viitor vârf selectat. Aceasta înseamnă că urmează şi alte iteraţii. Pentru orice descendent $d$ al lui $w$, $c^*(d) \leq u(d) \leq u(w) < u(w) + \varepsilon = u_{min}$. Astfel, toţi descendenţii lui $w$ vor fi introduşi în structura de aşteptare, deoarece $u_{min}$ rămâne neschimbat până la terminarea execuţiei algoritmului. Printre descendenţii lui $w$ se află şi vârfurile de pe frontiera arborelui cu rădăcina $w$. În concluzie, până la terminarea algoritmului, va fi inserat în structura de aşteptare un vârf de pe frontieră fără ca acesta să schimbe valoarea $u_{min}$, deoarece $w$ este ultimul care a făcut aceasta. □

**Teorema 12.2.** *Dacă $s$ este vârful calculat de* `BBCostMin&Marg()`, *atunci acesta este un vârf de pe frontieră pentru care $u(s) = \min\{u(v)\}$.*

*Demonstraţie.*

1. Deoarece $u(v) \geq c(v) \geq c^*(v)$, rezultă că dacă $c^*(v) \geq u_{min}$, atunci şi $u(v) \geq u_{min}$. Deci, este suficient să se testeze $c^*(v) < u_{min}$ pentru a decide dacă vârful $v$ va fi sau nu va fi inserat în structura de aşteptare. Aceasta conduce însă la inserarea în structura de aşteptare a unor vârfuri care pot avea $u(v) \geq u_{min}$, deoarece $c^*(v) < u_{min}$ nu asigură $u(v) < u_{min}$.

2. Atunci când un vârf de pe frontieră este inserat în structura de așteptare, acesta va avea costul $c^*(v) = c(v) = u(v)$ inferior lui $u_{min}$, deci costul său va defini noua valoare a lui $u_{min}$.
3. Vârfurile $v$ care schimbă valoarea lui $u_{min}$ și nu sunt vârfuri de pe frontieră, nu pot fi ultimele care realizează această modificare (lema 12.1).

Din 1-3 rezultă că ultimul vârf $s$ de pe frontieră reținut va fi și ultimul care micșorează $u_{min}$, iar alte vârfuri care pot micșora $u_{min}$ nu mai există. Astfel, $s$ satisface $u(s) = u = \min\{u(v)\}$. sfdem

## 12.4 Colorarea grafurilor

### 12.4.1 Descrierea problemei

Se consideră un (di)graf $G = (V, E)$ cu $V = \{1, \ldots, n\}$ și $m$ culori numerotate de la 1 la $m$. O *colorare* a (di)grafului este o funcție de la mulțimea vârfurilor $V$ la mulțimea culorilor $\{1, \ldots, m\}$ cu proprietatea că oricare muchie are extremitățile colorate diferit. Problema constă în generarea tuturor colorărilor posibile.

Problema este inspirată de la colorarea hărților. Fiecare hartă poate fi transformată într-un graf planar în modul următor: fiecare regiune a hărții este reprezentată printr-un vârf al grafului, iar două vârfuri sunt adiacente dacă regiunile corespunzătoare au o frontieră comună. De exemplu, harta din figura 12.10a este reprezentată de graful din figura 12.10b.

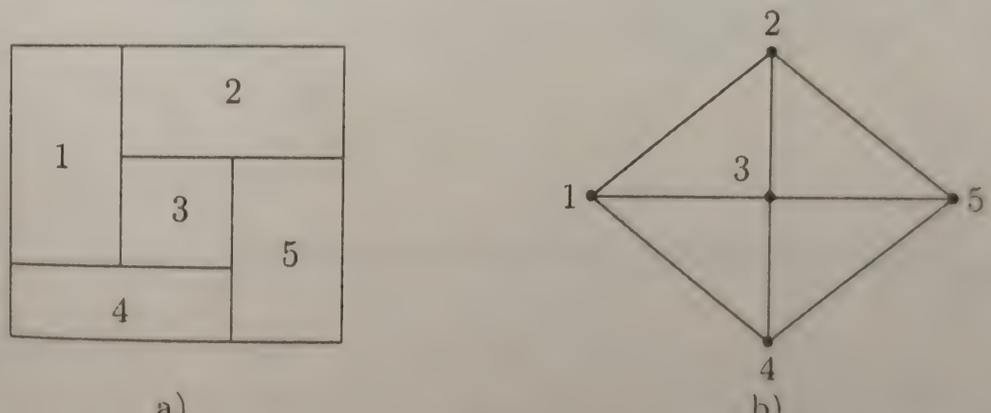
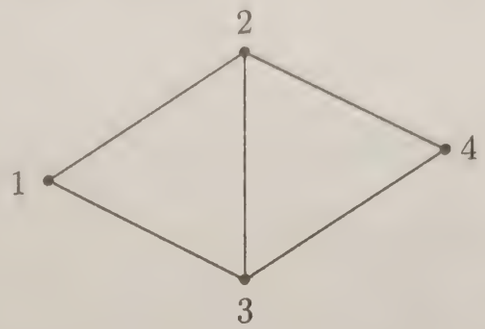



Figura 12.10: Relația de vecinătate dintr-o hartă reprezentată printr-un graf

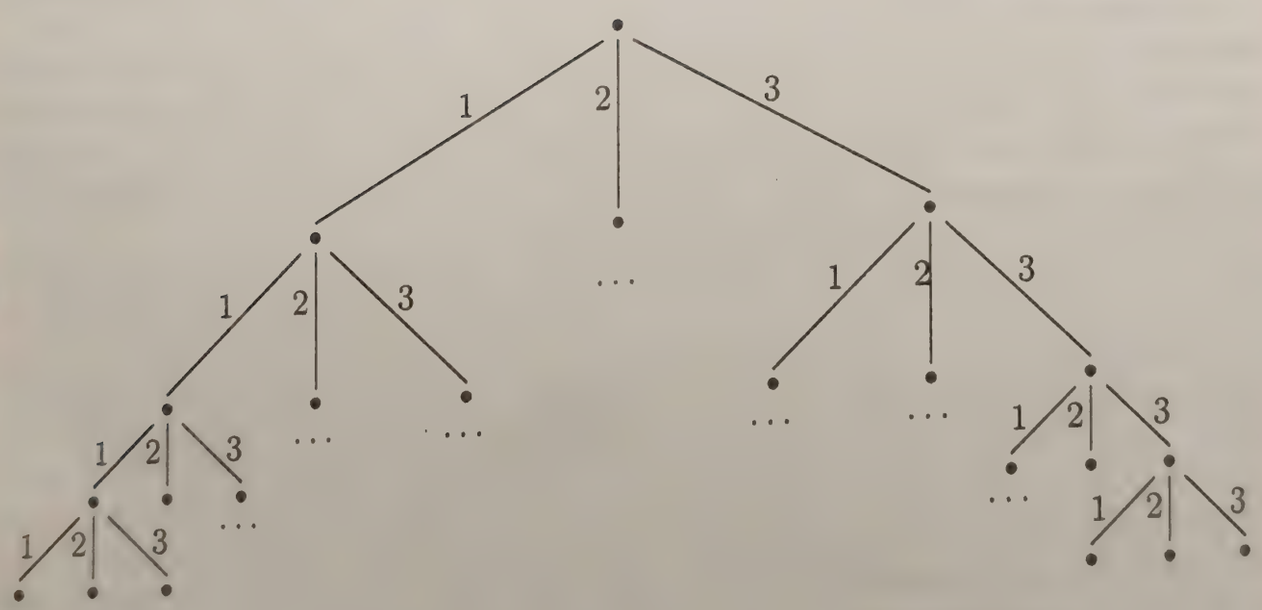
**Observație.** Pentru mulți ani s-a cunoscut faptul că cinci culori sunt suficiente pentru colorarea unei hărți, dar nu s-a găsit nici o hartă care să necesite mai mult de patru culori. După mai bine de o sută de ani (problema a fost formulată pentru prima dată în 1852 de către un student londonez, Francis Guthrie), s-a putut demonstra (Appel și Haken, 1976) că patru culori sunt suficiente pentru colorarea unei hărți. Demonstrația acestui rezultat include verificarea unei proprietăți de reductibilitate a grafurilor cu ajutorul calculatorului, fapt care a condus la „discuții aprinse" în ceea ce privește corectitudinea. sfobs

### 12.4.2 Modelul matematic

Convenim să notăm o soluţie a problemei printr-un vector $(x_1, \ldots, x_n)$ unde $x_i$ reprezintă culoarea vârfului $i$. De asemenea, presupunem că $G$ este graf. Astfel, spaţiul soluţiilor potenţiale este $\{1, \ldots, m\}^n$ şi poate fi reprezentat printr-un arbore cu $n$ nivele în care fiecare nod interior are exact $m$ succesori. De exemplu, graful din figura 12.11a are spaţiul soluţiilor potenţiale reprezentat de arborele din figura 12.11b.




a)



b)

Figura 12.11: Colorarea grafurilor – spaţiul soluţiilor potenţiale

Condiţia „nu există muchii cu extremităţile de aceeaşi culoare", prin care se stabileşte dacă un vector $(x_1, \ldots, x_n) \in \{1, \ldots, m\}^n$ este colorare, se exprimă prin:

$$\forall i, j \in V : \{i, j\} \in E \Rightarrow x_i \neq x_j \tag{12.1}$$

Criteriul de stabilire a unui candidat (de mărginire) se obţine restricţionând 12.1 la soluţia potenţială parţială $(x_1 \ldots, x_k)$:

$$\forall j : (1 \leq j \leq k-1) \wedge (\{j, k\} \in E \Rightarrow x_j \neq x_k) \tag{12.2}$$

Utilizând 12.2 ca funcţie de tăiere, arborele din exemplul de mai sus este transformat în arborele parţial din figura 12.12. Orice vârf de pe frontiera acestui arbore parţial este vârf soluţie.

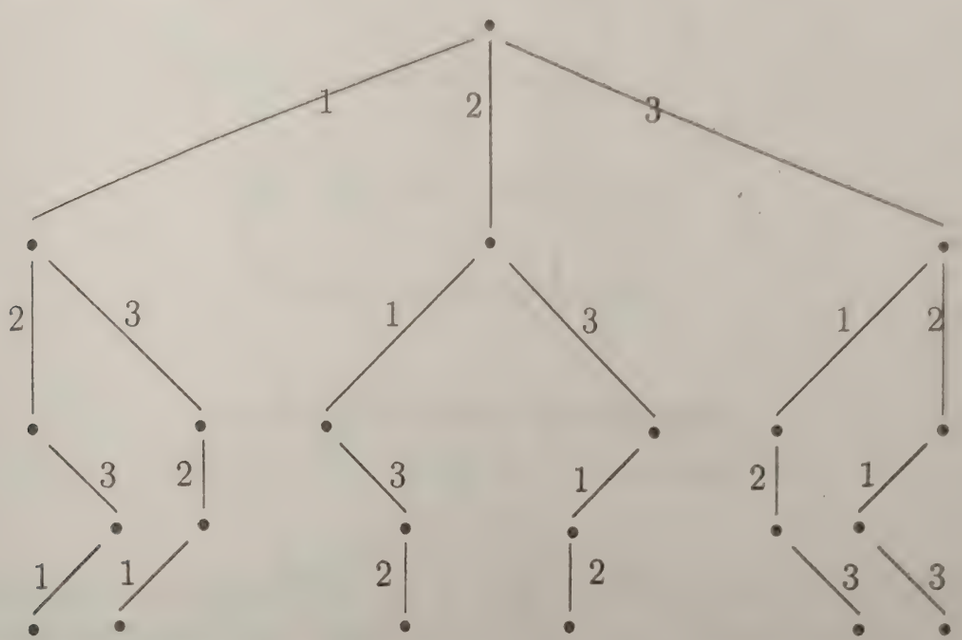



Figura 12.12: Colorarea grafurilor – arbore parţial

Ca algoritm de enumerare, utilizăm subprogramul de generare a elementelor produsului cartezian. Întrucât, de regulă, algoritmul de enumerare este determinat de structura de date aleasă pentru reprezentarea soluţiilor, rezultă că două sunt elementele ce caracterizează un algoritm backtracking: reprezentarea soluţiilor şi criteriul de mărginire.

### 12.4.3 Implementare

Pentru implementare considerăm grafurile reprezentate prin matricele de adiacenţă.

```
functia C(x, k)
   candidat ← true
   for j ← 1 to k-1 do
       if ((G.a[j,k]=1) and (x[j]=x[k]))
          then candidat ← false
   return candidat
end
```

## 12.5 Problema celor $n$ regine

### 12.5.1 Descrierea problemei

Se consideră o tablă de şah de dimensiune $n \times n$ şi $n$ regine. Problema constă în aşezarea pe tabla de şah a celor $n$ regine, astfel încât să nu se captureze una pe alta. Dacă există mai multe asemenea aşezări, atunci se vor determina toate.

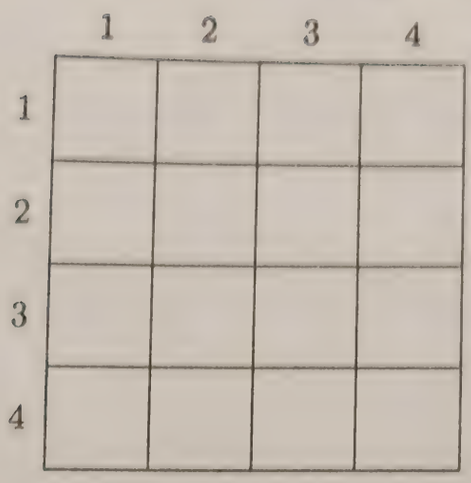



Figura 12.13: Tabla de şah $4 \times 4$

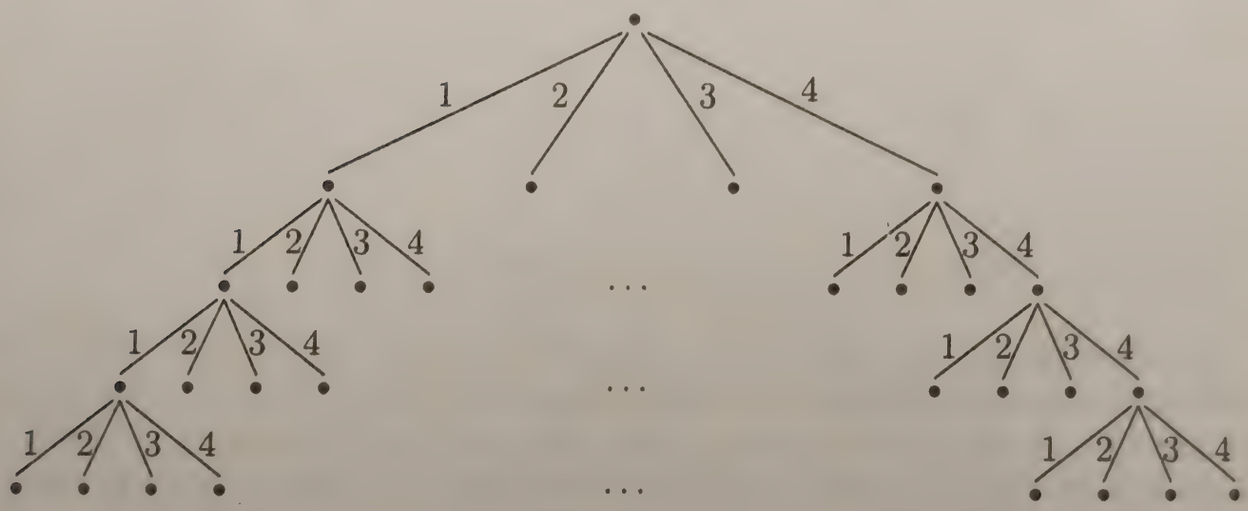



Figura 12.14: 4 regine – reprezentarea spaţiului soluţiilor potenţiale

### 12.5.2 Modelul matematic

Poziţiile de pe tabla de şah sunt identificate prin perechi $(i, j)$ cu $1 \leq i, j \leq n$. De exemplu, pentru $n = 4$, tabla este cea reprezentată în figura 12.13. O soluţie poate fi reprezentată printr-o funcţie $Q : \{1, \ldots, n\} \times \{1, \ldots n\} \rightarrow \{false, true\}$ cu semnificaţia: $Q[i, j] = true$ dacă şi numai dacă pe poziţia $(i, j)$ se află o regină. Se observă direct că spaţiul soluţiilor potenţiale conţine $2^{n^2}$ elemente. O reducere substanţială a acestuia se obţine dacă se reprezintă soluţiile printr-o funcţie $s : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ cu semnificaţia: $s[i] = j \Leftrightarrow Q[i, j] = true$. În acest caz, spaţiul soluţiilor potenţiale conţine $n^n$ elemente. De exemplu, pentru $n = 8$ avem $2^{n^2} = 2^{64}$, iar $8^8 = 2^{24}$. Aşadar alegerea reprezentării pentru soluţiile potenţiale este foarte importantă pentru metoda backtracking: o alegere bună poate conduce la redecerea substanţială a dimensiunii spaţiului acestor soluţii. Cele $n^n$ elemente ale spaţiului soluţiilor potenţiale pot fi reprezentate printr-un arbore cu $n$ nivele în care fiecare vârf are exact $n$ succesori imediaţi. Pentru cazul $n = 4$, arborele este cel reprezentat în figura 12.14.

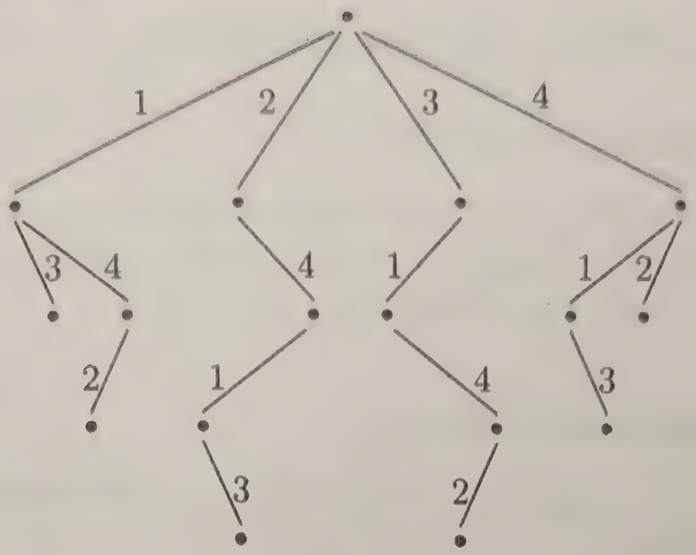



Figura 12.15: 4 regine – arborele parţial

În continuare vom căuta criteriul de mărginire prin care să eliminăm acei subarbori care nu conţin vârfuri-soluţie. Să observăm că $s_k$ candidează la soluţie, dacă aşezând cea de-a $k$-a regină pe poziţia $(k, s_k)$, ea nu este atacată de şi nu atacă nici una dintre cele $k-1$ regine aşezate pe liniile $1, \ldots, k-1$. Ecuaţiile celor patru direcţii pe care se poate deplasa o regină sunt date de:

regina de poziţia $(i, j)$ se poate deplasa pe poziţia $(k, \ell)$ dacă şi numai dacă:

- $i = k$ (deplasare pe orizontală) sau
- $j = \ell$ (deplasare pe verticală) sau
- $i - j = k - \ell$ (deplasare pe o diagonală principală) sau
- $i + j = k + \ell$ (deplasare pe o diagonală secundară).

Considerând $(i, j) = (i, s_i)$ şi $(k, \ell) = (k, s_k)$ obţinem condiţia care reprezintă criteriul de mărginire:

$$C(s_1, \ldots, s_k) = \forall i (1 \leq i \leq k-1): \neg Q[i, s_k] \wedge \neg(i + s_i = s_k +_i k \vee i - s_i = k - s_k))$$

Sau, ţinând cont de:

$$s_k + k = i + s_i \Rightarrow s_i = s_k + k - i \text{ şi } s_k - k = s_i - i \Rightarrow s_i = s_k - k + i$$

obţinem forma echivalentă:

$$C(s_1, \ldots, s_k) = \forall i (1 \leq i \leq k-1): \neg Q[i, s_k] \wedge \neg Q[i, s_k + k - i] \wedge \neg Q[i, s_k - k + i]$$

Pentru a ne face o idee cât de mult taie $C$ din arborele spaţiului soluţiilor potenţiale, prezentăm arborele parţial obţinut în urma aplicării criteriului de mărginire pentru cazul $n = 4$ în figura 12.15.

O funcţie $s : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ este în fapt un element din $\{1, 2, \ldots, n\}^n$ şi deci ca algoritm de enumerare poate fi folosită oricare dintre variantele de generare a elementelor produsului cartezian.

## 12.6 Submulţime de sumă dată

### 12.6.1 Descrierea problemei

Se consideră o mulţime $A$ cu $n$ elemente, fiecare element $a \in A$ având o dimensiune $s(a) \in \mathbb{Z}_+$ şi un număr întreg pozitiv $M$. Problema constă în determinarea tuturor submulţimilor $A' \subseteq A$ cu proprietatea $\sum_{a \in A'} s(a) = M$.

### 12.6.2 Modelul matematic

Presupunem $A = \{1, \ldots, n\}$ şi $s(i) = w_i, 1 \leq i \leq n$. Pentru reprezentarea soluţiilor avem două posibilităţi.

1. Prin vectori care să conţină elementele care compun soluţia. Această reprezentare are dezavantajul că trebuie utilizat un algoritm de enumerare a vectorilor de lungime variabilă. De asemenea testarea condiţiei $a \in A \setminus A'$? nu mai poate fi realizată în timpul $O(1)$ dacă nu se utilizează spaţiu de memorie suplimentar.
2. Prin vectori de lungime $n$, $(x_1, \ldots, x_n)$ cu $x_i \in \{0, 1\}$ având semnificaţia: $x_i = 1$ dacă şi numai dacă $w_i$ aparţine soluţiei (vectorii caracteristici).

**Exemplu.** Fie $n = 4$, $(w_1, w_2, w_3, w_4) = (4, 7, 11, 14)$ şi $M = 25$. Există următoarele soluţii:

- $(4, 7, 14)$ care mai poate fi reprezentată prin $(1, 2, 4)$ sau $(1, 1, 0, 1)$ şi
- $(11, 14)$ care mai poate fi reprezentată prin $(3, 4)$ sau $(0, 0, 1, 1)$.

sfex

Noi vom opta pentru ultima variantă, deoarece vectorii au lungime fixă. Remarcăm faptul că spaţiul soluţiilor conţine $2^n$ posibilităţi (elementele mulţimii $\{0, 1\}^n$) şi poate fi reprezentat printr-un arbore binar. Ca algoritm de enumerare vom utiliza `GenProdCart`. Aşa cum am mai văzut, acesta generează soluţiile potenţiale prin partiţionarea mulţimii $A$ în două: o parte $\{1, \ldots, k\}$ care a fost luată în considerare pentru a stabili candidaţii la soluţie şi a doua parte $\{k + 1, \ldots, n\}$ ce urmează a fi luată în considerare. Cele două părţi trebuie să satisfacă următoarele două inegalităţi:

- suma parţială dată de prima parte (adică de candidaţii aleşi) să nu depăşească $M$:

$$\sum_{i=1}^{k} x_i \cdot w_i \leq M \tag{12.3}$$

- ceea ce rămâne să fie suficient pentru a forma suma $M$:

$$\sum_{i=1}^{k} x_i \cdot w_i + \sum_{i=k+1}^{n} w_i \geq M \tag{12.4}$$

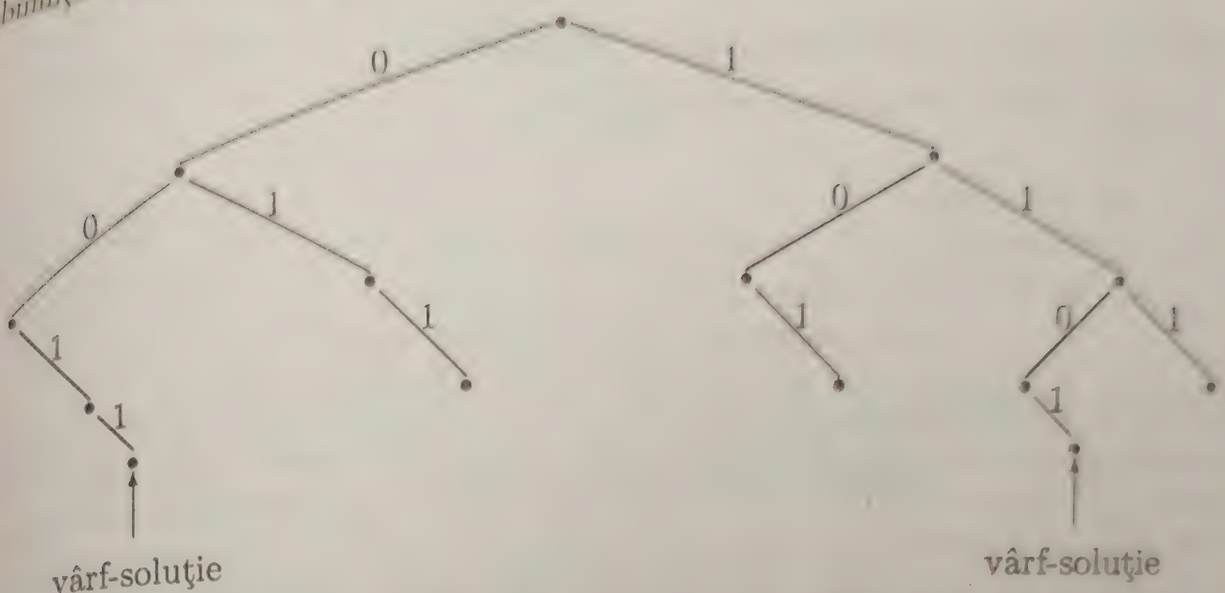



Figura 12.16: Arbore parţial pentru submulţime de sumă dată

Cele două inegalităţi pot constitui criteriul de mărginire. Cu acest criteriu de tăiere, arborele parţial rezultat pentru exemplul anterior este cel reprezentat în figura 12.16.

De remarcat că acest criteriu nu elimină toţi subarborii care nu conţin vârfuri-soluţie, dar elimină foarte mulţi, restrângând astfel spaţiul de căutare. Atingerea unui vârf pe frontieră presupune imediat determinarea unei soluţii: suma $\sum_{i=k-1}^{n} w_i$ este zero (deoarece $k = n$) şi dubla inegalitate dată de relaţiile 12.3 şi 12.4 implică $\sum_{i=1}^{n} w_i = M$.

**Observaţie.** Dacă se utilizează drept criteriu de mărginire numai inegalitatea 12.3, atunci atingerea unui vârf pe frontieră în arborele parţial nu presupune neapărat şi obţinerea unei soluţii. Mai trebuie verificat dacă suma submulţimii alese este exact $M$.

sfobs

### 12.6.3 Optimizare

O varianta optimizată a algoritmului *backtracking* pentru problema submulţimilor de sumă dată se obţine astfel:

- se consideră $w_1, w_2, ..., w_n$ în ordine crescătoare (fără a restrânge generalitatea);
- pentru cazul în care suma parţială dată de candidaţii aleşi este strict mai mică decât $M$ ($\sum_{i=1}^{k} x_i w_i < M$), se introduce un criteriu de mărginire suplimentar

$$\sum_{i=1}^{k} x_i w_i + w_{k+1} \leq M.$$

Presupunem valorile $x_1, ..., x_{k-1}$ calculate. Notăm cu $s$ suma parţială corespunzătoare valorilor $x_1, ..., x_{k-1}$ ($s = \sum_{i=1}^{k-1} x_i w_i$) şi cu $r$ suma $\sum_{i=k}^{n} w_i$. Presupunem $w_1 \leq M$ şi $\sum_{i=1}^{n} w_i \geq M$.

Se obţine astfel următorul algoritm *backtracking*:

```
procedure submultimiOpt(s,k,r)
 x_k ← 1
 if (s+w_k=M)
   then scrie(x^k) /* x^k=(x_1,x_2,...,x_k)) */
   else if (s+w_k+w_{k+1} ≤ M)
           then submultimiOpt(s+w_k,k+1,r-w_k)
        if ((s+r-w_k ≥ M) and (s+w_{k+1} ≤ M))
           then x_k ← 0
           submultimiOpt(s,k+1,r-w_k)
end
```

Apelul iniţial este $\texttt{submultimiOpt}\left(0, 1, \sum_{i=1}^{n} w_i\right)$, iar $w_1 \leq M$ şi $0 + \sum_{i=1}^{n} w_i \geq M$. Astfel, la intrarea în execuţie sunt asigurate condiţiile $s + w_k \leq M$ şi $s + r \geq M$. Aceste condiţii sunt asigurate şi la apelul recursiv: înainte de primul apel recursiv, $\texttt{submultimiOpt}(s + w_k, k + 1, r - w_k)$, nu e nevoie să se mai verifice dacă $\sum_{i=1}^{k} x_i w_i + \sum_{i=k+1}^{n} w_i \geq M$, deoarece $s + r > M$ şi $x_k = 1$.

Nu se verifică explicit nici $k > n$. Iniţial, $s = 0 < M$ şi $s + r \geq M$. Rezultă $r \neq 0$, deci $k$ nu poate depăşi $n$. De asemenea, în linia „if $(s + w_k + w_{k+1} \leq M)$", deoarece $s + w_k < M$, rezultă $r \neq w_k$, deci $k + 1 \leq n$.

## 12.7 Problema rucsacului II (continuare)

### 12.7.1 Descriera problemei

Reamintim problema rucsacului în varianta discretă, descrisă la capitolul programare dinamică:

> Se consideră $n$ obiecte, $1, \ldots, n$, de dimensiuni (greutăţi) $w_1, \ldots, w_n \in \mathbb{Z}_+$ şi un rucsac de capacitate $M \in \mathbb{Z}_+$. Un obiect $i$ sau este introdus în totalitate în rucsac, $x_i = 1$, sau nu este introdus de loc, $x_i = 0$, astfel că o umplere a rucsacului constă dintr-o secvenţă $x_1, \ldots, x_n$ cu $x_i \in \{0, 1\}$ şi $\sum_i x_i \cdot w_i \leq M$. Introducerea obiectului $i$ în rucsac aduce profitul $p_i \in \mathbb{Z}$, iar profitul total este $\sum_{i=1}^{n} x_i p_i$. Problema constă în a determina o alegere $(x_1, \ldots, x_n)$ care să aducă un profit maxim.

### 12.7.2 Modelul matematic

Convenim să notăm cu $(y_1, \ldots, y_n)$ soluţia curentă. Presupunem că au fost selectate valorile $y_i, 1 \leq i \leq k$, şi urmează să determinăm dacă $y_k$ poate candida la o soluţie optimă. Obţinem un criteriu de mărginire bun dacă dispunem de o margine superioară pentru valoarea celei mai bune soluţii ce se poate obţine pentru alegerea făcută până la momentul considerat. Această margine poate fi obţinută dacă relaxăm problema în modul următor:

- funcţia obiectiv:

$$\max \sum_{i=1}^{n} y_i \cdot p_i$$

- restricţii:

$$y_i \in \{0,1\}, 1 \leq i \leq k$$
$$0 \leq y_i \leq 1, k+1 \leq i \leq n$$
$$\sum_{i=1}^{n} y_i \cdot w_i \leq M$$

Reamintim că valorile pentru $y_i$ cu $1 \leq i \leq k$ sunt deja atribuite. Se rezolvă problema relaxată utilizând algoritmul greedy, obţinându-se o margine superioară $fsup$ (profitul maxim care se poate realiza cu $(y_1, \ldots, y_k)$). Fie $x_i \in \{0,1\}, 1 \leq i \leq n$, o soluţie arbitrară. De fapt, vom vedea că $(x_i)_{1 \leq i \leq n}$ va reprezenta cea mai bună soluţie determinată până la momentul curent. Dacă:

$$\sum_{i=1}^{n} x_i \cdot p_i \geq fsup$$

atunci, evident, $(y_1, \ldots, y_k)$ nu poate candida la o soluţie optimă şi trebuie făcută o altă alegere pentru primele $k$ componente. Această nouă alegere se face în maniera următoare:

- se determină cel mai mare $k'$ cu $k' \leq k$ şi $y'_k = 1$ ($k'$ este ultimul obiect introdus în rucsac);
- se face $y'_k = 0$ (se scoate ultimul obiect introdus) şi se testează dacă $(y_1, \ldots, y_{k'})$ poate candida la o soluţie optimă.

Acum, ordinea atribuirii de valori pentru $y_k$ este următoarea: întâi se consideră $y_k$ egal cu 1 (se încearcă mai întâi introducerea obiectului $k$ în rucsac). Se testează dacă această alegere poate conduce la o soluţie mai bună. Dacă da, atunci se determină cea mai bună soluţie pentru alegerea făcută până în acel moment şi se actualizează. Dacă această alegere nu poate conduce la o soluţie mai bună decât cea întâlnită până în acel moment, atunci se atribuie lui $y_k$ valoarea zero (obiectul $k$ nu va fi introdus în rucsac).

Criteriul de mărginire se bazează pe cunoaşterea unei soluţii $(x_1, \ldots, x_n)$ care să reprezinte optimul peste soluţiile potenţiale explorate. Valorile iniţiale pentru $x_i$ pot fi alese cu un algoritm greedy: se alege obiectul ce aduce un profit maxim peste mulţimea obiectelor nealese. Presupunem $\frac{p_1}{w_1} \geq \cdots \geq \frac{p_n}{w_n}$.

Determinarea soluţiei iniţiale $(x_1, \ldots, x_n)$ coincide cu parcurgerea primelor $n$ vârfuri din explorarea DFS a arborelui corespunzător spaţiului soluţiilor potenţiale. Pentru a nu mai calcula încă o dată această soluţie, se consideră iniţial $(y_1, \ldots, y_n) = (x_1, \ldots, x_n)$.

## 12.7.3 Implementare

### 12.7.3.1 *Backtracking*

Vom considera un subprogram care determină soluţia iniţială:

```
procedure detSolInit(ob, n, M, k, y, pinit, cinit)
   cinit ← 0
   pinit ← 0
   i ← 1
   while (i ≤ n) do
      cinit ← cinit+ob[i].w
      if (cinit <= M)
         then ob[i].x ← 1
              y[i] ← 1
              pinit ← pinit+ob[i].p
              i ← i+1
         else k ← i-1
              cinit ← cinit-ob[i].w
              i ← n+1
end
```

Procedura care calculează problema relaxată are o descriere asemănătoare:

```
function fSup(ob, n, M, k, ck, pk)
   ptemp ← pk
   ctemp ← ck
   for i ← k+1 to n do
       ctemp ← ctemp+ob[i].w
       if (ctemp < M)
          then ptemp ← ptemp+ob[i].p
          else ptemp ← ptemp+(1-(ctemp-M)/ob[i].w)*ob[i].p
               return ptemp
   return ptemp
end
```

Algoritmul backtracking care rezolvă problema rucsacului este descris de următorul program:

```
procedure rucsacIIBack(ob, n, M, copt, popt)
   detSolInit(ob, n, M, k, y, ptemp, ctemp)
   popt ← ptemp
   while (k > 0) do
      while (fsup(ob, n, k, M, ctemp, ptemp) ≤ popt) do
         while ((k > 0) and (y[k] = 0)) do
            k ← k-1
         if (k > 0)
            then y[k] ← 0
                 ctemp ← ctemp-ob[k].w
                 ptemp ← ptemp-ob[k].p
                 while ((k < n) and (ob[k+1].w+ctemp ≤ M)) do
                    k ← k+1
                    y[k] ← 1
                    ctemp ← ctemp+ob[k].w
                    ptemp ← ptemp+ob[k].p
```

```
        if (k=n)
           then popt ← ptemp
                copt ← ctemp
                for i ← 1 to n do
                    ob[i].x ← y[i]
           else k ← k+1
                y[k] ← 0
end
```

**Exemplu.** Presupunem $n = 4$ şi greutăţile şi profiturile date de următorul tabel:

| $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $p_i$ | 40 | 50 | 55 | 70 |
| $w_i$ | 30 | 40 | 45 | 60 |

Spaţiul soluţiilor potenţiale este reprezentat de arborele din figura 12.17. Pentru vârfurile de pe frontieră sunt trecute greutăţile şi profiturile corespunzătoare soluţiilor date de soluţiile potenţiale date de aceste vârfuri. Considerăm două cazuri:

1. $M = 100$. Arborele parţial parcurs de algoritmul backtracking este cel din figura 12.18. Calculele corespunzătoare vârfurilor $A, B, C, D$, alese la întâmplare, sunt:

$$A: \quad fsup = 40 + 55 + \frac{25}{60} \cdot 70 > popt = 90$$
$$B: \quad fsup = 40 + 70 = 110 > popt = 95$$
$$C: \quad fsup = 50 + 55 + \frac{15}{60} \cdot 70 > popt = 110$$
$$D: \quad fsup = 55 + \frac{55}{60} \cdot 70 = \frac{330 + 385}{6} = \frac{715}{6} < popt = 120$$

2. $M = 80$. Arborele parţial parcurs de algoritmul backtracking este cel din figura 12.19. Calculele corespunzătoare vârfurilor $A, B, C, D$ sunt:

$$A: \quad fsup = 40 + 55 + \frac{5}{60} \cdot 70 > popt = 95$$
$$B: \quad fsup = 40 + \frac{50}{60} \cdot 70 = \frac{240 + 350}{6} = \frac{590}{6} > popt = 95$$
$$C: \quad fsup = 50 + \frac{40}{45} \cdot 85 = \frac{450 + 40}{4} = \frac{890}{9} > popt = 95$$
$$D: \quad fsup = 55 + \frac{35}{60} \cdot 70 = \frac{330 + 245}{6} = \frac{575}{7} < popt = 95$$

sfex

### 12.7.3.2 *Branch and bound* cu cost minim şi mărginire

Problema rucsacului implică maximizarea funcţiei profit. Pentru a fi posibilă aplicarea strategiei cu cost minim şi mărginire, funcţia obiectiv $\max \sum_{i=1}^{n} x_i \cdot p_i$ este transformată în $\min \sum_{i=1}^{n} x_i \cdot (-p_i)$.

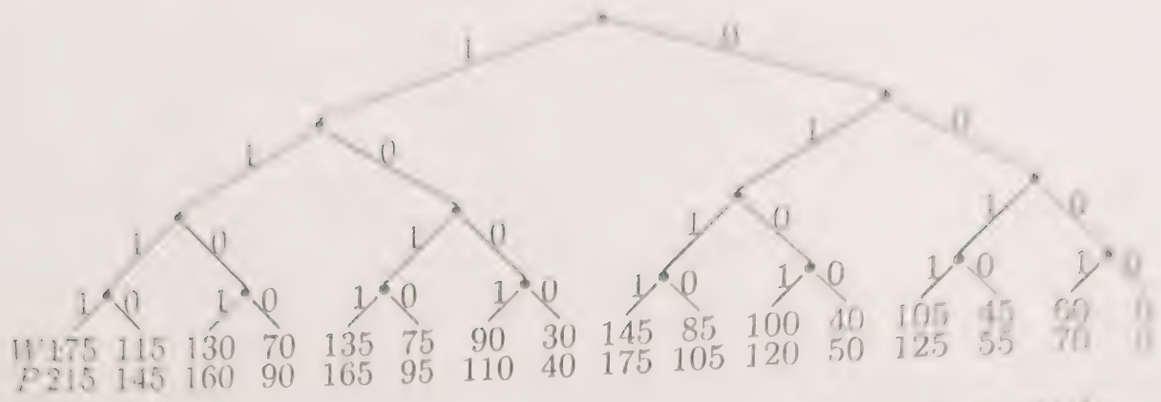



Figura 12.17: Problema rucsacului – arborele soluţiilor potenţiale

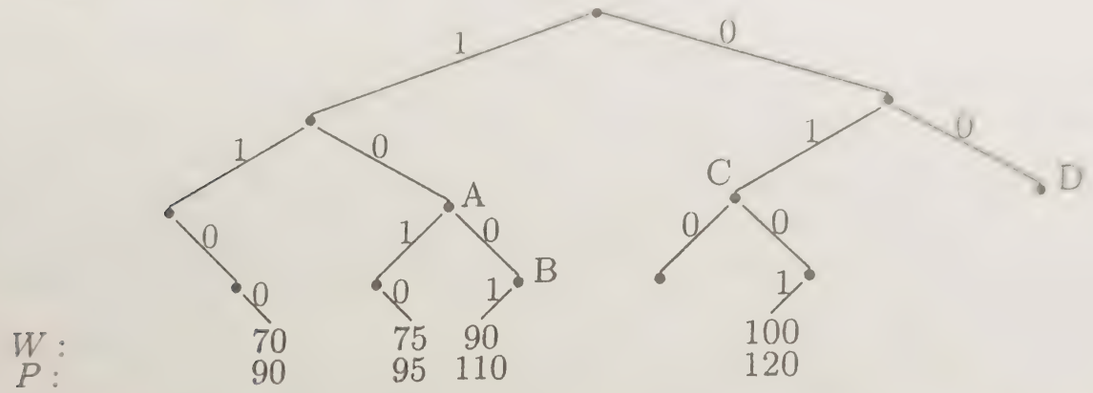



Figura 12.18: Problema rucasacului – arborele parţial pentru $M = 100$

În aceste condiţii, pentru un vârf $v$, $c^*(v)$ se defineşte ca *–profitul maxim posibil* pentru $v$ şi $u(v)$ ca *–profitul sigur.*

Considerăm obiectele ordonate astfel încât $\frac{p_1}{w_1} \geq \cdots \geq \frac{p_n}{w_n}$. Fie $v$ un vârf de pe nivelul $k$; valorile $x_1, x_2, ..., x_{k-1}$ sunt deja calculate. Notăm cu $c_r$ capacitatea rămasă şi cu $p_o$ profitul deja obţinut. Presupunem că obiectele $k, \ldots, n$ nu au fost încă analizate. În aceste condiţii, valorile $c^*(v)$ şi $u(v)$ pot fi calculate cu procedura următoare:

```
procedure margini(p, w, cr, po, n, k, c*, u)
 u ← -po; cap ← cr;
 for i ← k to n do
   if (cap ≥ w[i])
      then cap ← cap-w[i] /* obiectul i poate fi incărcat în rucsac */
           u ← u-p[i]
      else c* ← u - cap * (p[i]/w[i])
           for j ← i+1 to n do
               if(cap ≥ w[j])
                  then cap ← cap-w[j]/* obiectul j are loc în rucsac */
                       u ← u-p[j]
           return
 c* ← u /* toate obiectele k, k+1, ..., n pot fi încărcate în rucsac */
end
```

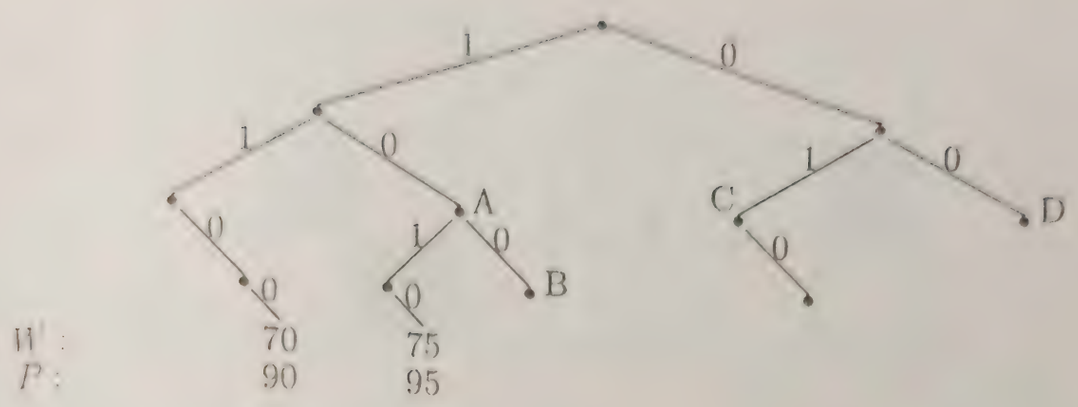



Figura 12.19: Problema rucasacului: arborele parţial pentru $M = 80$

Observaţie. Dacă $x_k = 1$, adică `cap` $\geq$ `w(k)`, atunci valorile `c*` şi `u` calculate în urma apelurilor:

```
margini(p, w, c_r, p_o, n, k, c*, u)
```

şi

```
margini(p, w, c_r - w[i], p_o + p[k], n, k + 1, c*, u)
```

sunt identice. sfobs

Pentru rezolvarea problemei prin metoda *branch-and-bound* cu cost minim şi mărginire, se poate aplica algoritmul standard `BBCostMin&Marg()`, care să folosească valorile $c^*(v)$ şi $u(v)$ calculate cu procedura `margini`.

## 12.8 *Perspico*

### 12.8.1 Descrierea problemei

Considerăm următoarea variantă simplificată a jocului *Perspico*. Fie o reţea formată din 3×3 pătrate, numite *locaţii*. În primele opt locaţii se găsesc opt piese etichetate cu litere de la A la H, de dimensiuni potrivite astfel încât să poată fi deplasate pe orizontală sau verticală atunci când există o locaţie vecină liberă (a se vedea figura 12.20). Numim această aşezare a pieselor *configuraţie finală* şi o notăm cu $C_f$. O *mutare* constă în deplasarea unei piese în locaţia liberă, atunci când este posibil.

Problema este următoarea:

1. dată o configuraţie $C$, să se decidă dacă există o listă de mutări care să permită obţinerea lui $C_f$ din $C$;
2. în cazul în care există, să se determine o astfel de listă de mutări.

### 12.8.2 Modelul matematic

Spaţiul soluţiilor conţine 9! combinaţii ($\infty$ dacă nu se verifică ciclităţile). În plus, interesează modul de obţinere a soluţiilor, adică drumul de la configuraţia $C$ la

Figura 12.20: Jocul *Perspico*

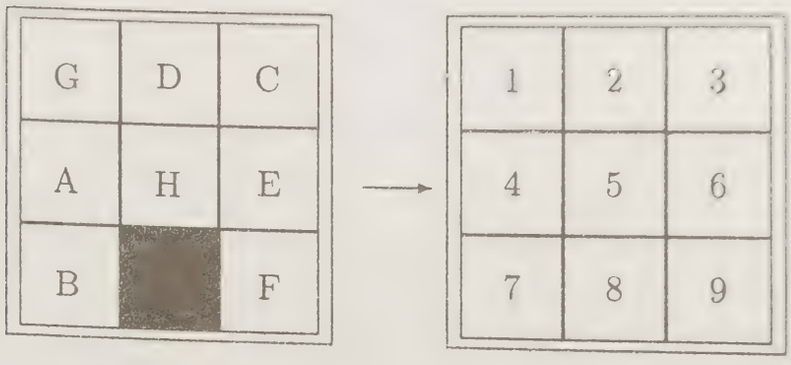



Figura 12.21: Configuraţia definită prin funcţia *poz*

configuraţia $C_f$.

Dată o configuraţie $C$, se poate decide dacă există o listă de mutări care să permită obţinerea lui $C_f$ din $C$. Dată o configuraţie $C$, se defineşte funcţia injectivă:

$$poz_C : \{\text{A,B,C,...,L}\} \to \{1, 2, 3, ..., 8, 9\}$$

unde $poz_C(X) = i$ semnifică faptul că piesa notată cu $X$ se află pe poziţia $i$. Locaţia liberă a fost asociată cu piesa simbolică L. În figura 12.21, $poz_C(\text{D}) = 2$, $poz_C(\text{L}) = 8$.

Pentru o configuraţie $C$ şi pentru fiecare piesă $X$ se defineşte

$$less_C(X) = \#\{Y | \text{A} \leq Y \leq \text{L}, Y < X, poz_C(Y) > poz_C(X)\}.$$

Peste mulţimea $\{\text{A}, \text{B}, \text{C}, ..., \text{H}, \text{L}\}$, se consideră ordinea alfabetică. În exemplul anterior avem $less_C(\text{H}) = 3$, $less_C(\text{L}) = 1$.

**Teorema 12.3.** *Fie $C$ o configuraţie, astfel încât $i, j \in \{1, 2, 3\}$ desemnează linia şi coloana unde este plasată locaţia liberă, piesa L, şi*

$$l(C) = \begin{cases} 0, & \text{dacă } i + j \text{ este par;} \\ 1, & \text{dacă } i + j \text{ este impar.} \end{cases}$$

*Atunci nu există un şir de transformări până la configuraţia finală $C_f$ dacă:*

$$S(C) = \sum_{X=\text{A}}^{\text{L}} less_C(X) + l(C) \quad \text{este număr impar.}$$

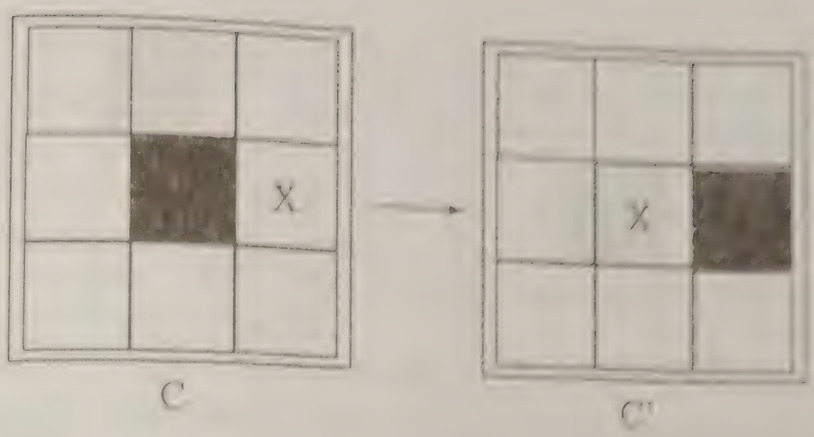



Figura 12.22: Locaţia liberă se deplasează la dreapta

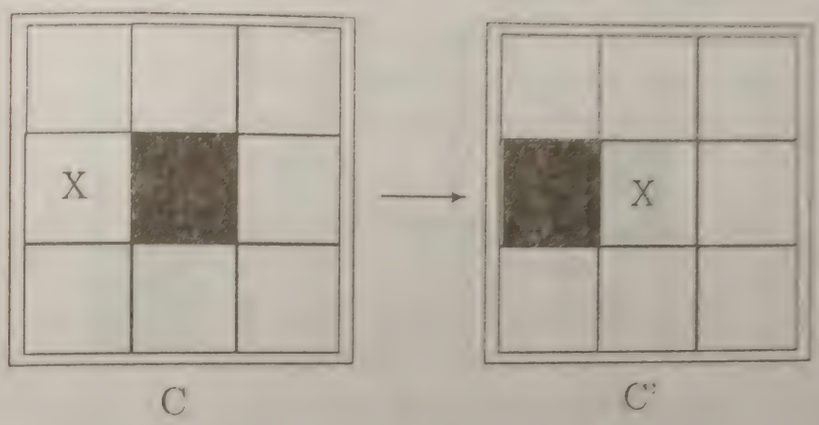



Figura 12.23: Locaţia liberă se deplasează la stânga

*Demonstraţie.* Pentru configuraţia finală $C_f$ : $S(C_f) = 0 + 0$, care este evident par. Fie o configuraţie $C$ pentru care $S(C) =$ este număr par şi $C'$ obţinută din $C$ prin una din mutările permise.

- Cazul 1. Locaţia liberă se deplasează la dreapta (figura 12.22). Avem: $l(C') = (l(C)+1) \bmod 2$, $less_{C'}(\mathrm{L}) = less_C(\mathrm{L}) - 1$, $less_{C'}(X) = less_C(X)$, $(\forall) X \neq \mathrm{L}$. Rezultă $S(C')$ număr par.
- Cazul 2. Locaţia liberă se deplasează la stânga (figura 12.23). Similar cu cazul 1.
- Cazul 3. Locaţia liberă se deplasează în sus (fig. 12.24). Avem:
  3.1 $l(C') = (l(C) - 1) \bmod 2$;
  3.2 $less_{C'}(\mathrm{L}) = less_C(\mathrm{L}) + 3$;
  3.3 $X$ schimbă locul cu $L$. Dacă $X < Y$, atunci $less_{C'}(Y) = less_C(Y) + 1$ şi $less_{C'}(X) = less_C(X)$; dacă $X > Y$, atunci $less_{C'}(X) = less_C(X) - 1$ şi $less_{C'}(Y) = less_C(Y)$. Analiza de mai sus este valabilă şi pentru $Z$, deci la $S(C)$ se adună un număr par (-2, 0 sau 2).
  Combinând cu 3.1, 3.3 şi 3.3, rezultă că $S(C')$ rămâne par.
- Cazul 4. Locaţia liberă se deplasează în jos (figura 12.25):
  Simetric cu cazul 3.

Deoarece prin transformările $\rightarrow\uparrow\downarrow\leftarrow$ nu se schimbă paritatea valorii $S(C)$ şi cum $S(C_f)$ este 0 (par), rezultă că nu există un şir de transformări până la configuraţia finală $C_f$, dacă $S(C)$ este număr impar. sfdem

Figura 12.24: Locaţia liberă se deplasează în sus

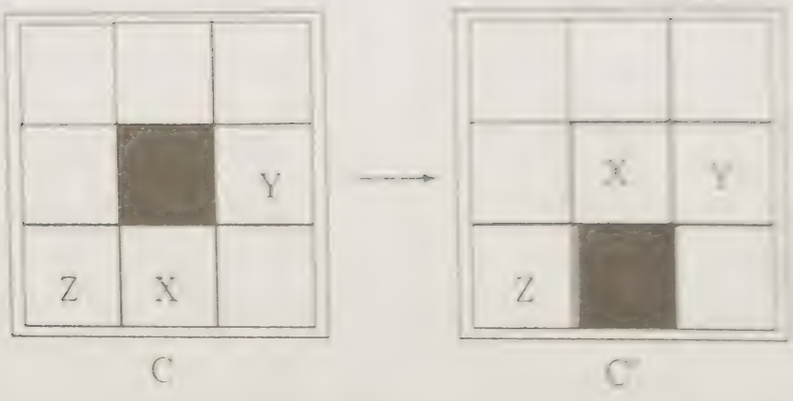



Figura 12.25: Locaţia liberă se deplasează în jos

### 12.8.3 Implementare

Pentru a rezolva problema determinării listei de mutări care să permită obţinerea lui $C_f$ din $C$, dacă această există, poate fi aplicat algoritmul branch-and-bound cu cost minim.

În acest caz $c^*(x) = f(x) + g^*(x)$, unde: $f(x) = \text{nivel}(x)$, iar $g^*(x)$ = numărul pieselor care nu sunt pe poziţiile definite de $C_f$.

## 12.9 Exerciţii

**Exerciţiul 12.1.** Să se proiecteze un algoritm backtracking pentru problema din exerciţiul 9.2 (*Schimbarea banilor*).

**Exerciţiul 12.2.** (*Problema labirintului*) Se consideră un tablou bidimensional cu elemente 0 şi 1 reprezentînd un labirint: 1 blochează calea, iar 0 semnifică o poziţie deschisă. Să se scrie un algoritm backtracking care încearcă să traseze un drum de la poziţia $(1, 1)$ la poziţia $(n, n)$.

**Exerciţiul 12.3.** [HS84] (*Mărcile poştale*) Se consideră $n$ denumiri de mărci poştale şi se presupune că nu se permit mai mult de $m$ mărci poştale pe o scrisoare. Să se scrie un algoritm backtracking care determină cel mai mare domeniu (mulţime de numere consecutive) de valori poştale ce pot fi puse pe o scrisoare şi toate combinaţiile posibile de denumiri care realizează domeniul.

**Exerciţiul 12.4.** [HS84] (*Conectarea tranzistorilor*) Se consideră:

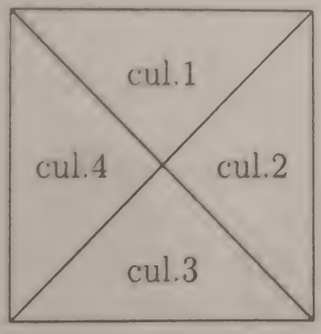
- $n$ componente electrice (tranzistori) ce urmează a fi plasaţi pe o placă de circuite pe care sunt marcate $n$ poziţii;
- o matrice de conectare $CON$ cu $CON[i,j]$ reprezentând numărul de conectări ce trebuie făcute între componentele $i$ şi $j$;
- o matrice de distanţe $DIST$ cu $DIST[r,s]$ reprezentând distanţa de la poziţia $r$ la poziţia $s$ a plăcii de circuite;
- costul unei conectări este $\sum_{i,j} CON[i,j] \cdot DIST[i,j]$.

Să se scrie un algoritm backtracking care determină o plasare a componentelor pe poziţiile plăcii, astfel încât costul de conectare să fie minim.

**Exerciţiul 12.5.** [HS84] (*Procesare paralelă*) Există $n$ programe ce urmează a fi executate de $k$ procesoare care lucrează în paralel. Timpul necesar executării programului $i$ este $t_i$. Scrieţi un algoritm backtracking care să determine ce programe să fie executate de fiecare procesor şi în ce ordine, astfel încât timpul după ce se termină ultimul program să fie minim.

**Exerciţiul 12.6.** (*Mutarea calului*) Se consideră o tablă de şah $n \times n$ pe care se plasează arbitrar un cal pe poziţia $(x, y)$. Să se determine $n^2 - 1$ mutări ale calului, astfel încât fiecare poziţie a tablei să fie vizitată de către acesta exact o dată. Există asemenea secvenţe de mutări pentru orice $n$?

**Exerciţiul 12.7.** [MS91] (*Mozaic*) O plăcuţă de mozaic are formă pătrată de dimensiune 1 şi este divizată în patru părţi colorate diferit. Fiecare parte colorată este unic determinată de o latură şi centrul pătratului.




Există $m$ tipuri de plăcuţe, *i.e.*, $m$ aranjamente coloristice diferite. De fiecare tip există $t_i$ plăcuţe, astfel încât $\sum_{i=1}^{m} t_i = n^2$. Să se scrie un algoritm *backtracking* care să determine o amplasare a plăcuţelor într-un pătrat de latura $n$, astfel încât ori de câte ori două plăcuţe au o latură comună să existe aceeaşi culoare de o parte şi de alta a laturii comune. Să se considere şi cazul când pe o plăcuţă pot să fie mai puţin de patru culori distincte.

**Exerciţiul 12.8.** [MS91] (*Galeriile de artă*) Să se scrie un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vârfuri, astfel încât reuniunea câmpurilor interne de vizibilitate acoperă tot interiorul poligonului.

**Exerciţiul 12.9.** [MS91] (*Fortăreaţa*) Să se scrie un un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vârfuri, astfel încât reuniunea câmpurilor externe de vizibilitate acoperă tot exteriorul poligonului.

**Exercițiul 12.10.** [MS91] (*Curtea închisorii*) Să se scrie un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vârfuri astfel încât reuniunea câmpurilor interne și externe de vizibilitate acoperă tot planul.

**Exercițiul 12.11.** Să se arate că orice număr natural $n > 2$ se poate scrie ca o sumă de numere prime. Să se scrie un program care, pentru un număr natural $n > 2$ dat, determină o secvență de lungime minimă de numere prime a căror sumă este egală cu $n$.

## 12.10 Referințe bibliografice

Acest capitol are la bază [HS84, Sah98, LG86]. In [Luc93] este descris un algoritm backtracking pentru rezolvarea ecuațiilor diofantice liniare.

# Capitolul 13

# Probleme $\mathbb{NP}$-complete

## 13.1 Algoritmi nedeterminişti

Activitatea unui algoritm nedeterminist se desfăşoară în două etape: într-o primă etapă „se ghiceşte" o anumită structură $S$ şi în etapa a doua se verifică dacă $S$ satisface o condiţia de rezolvare a problemei. Putem adăuga „puteri magice de ghicire" unui limbaj de programare adăugându-i o funcţie de forma:

`choice(M)` – care întoarce un element din mulţimea $M$.

Pentru a şti dacă verificarea s-a terminat cu succes sau nu, se adăugă şi două instrucţiuni de terminare:

`success` - care semnalează terminarea verificării (şi a algoritmului) cu succes, şi

`failure` - care semnalează terminarea verificării (şi a algoritmului) fără succes.

Această definiţie a algoritmilor nedeterminişti este strâns legată de rezolvarea problemelor de decizie, ce constituie forma standard utilizată de teoria calculabilităţii. Alegerea pare oarecum surprinzătoare având în vedere că foarte multe probleme studiate sunt probleme de optimizare. Cum pot fi formalizate problemele ca probleme de decizie? Prezentăm ca exemplu problema rucsacului, varianta discretă:

**Rucsac 0/1**

*Instanţă.* O mulţime $O$ (obiectele), o mărime $s(o) \in \mathbb{Z}_+$ şi o valoare $v(o) \in \mathbb{Z}_+$ pentru fiecare obiect $o \in O$, o restricţie $M \in \mathbb{Z}_+$ şi un scop $K \in \mathbb{Z}_+$.

*Întrebare.* Există o submulţime $O' \subseteq O$ cu proprietatea

$$\sum_{o \in O'} s(o) \leq M \text{ şi } \sum_{o \in O'} v(o) \geq K?$$

Se poate dovedi că problema de optim poate fi redusă polinomial la problema de decizie şi reciproc (vom vedea în secţiunea următoare ce înseamnă exact acest lucru). În acest fel, cele două probleme sunt echivalente din punctul de vedere al calculabilităţii.

Un algoritm nedeterminist care rezolvă problema de decizie a rucsacului este următorul:

```
procedure rucsacIIBack(O, s, v, n, M, K, x)
   /* etapa de ghicire */
   for fiecare o ∈ O do
      x[o] ← choice(0,1)
   /* etapa de verificare */
   sGhicit ← 0
   vGhicit ← 0
   for fiecare o ∈ O do
      sGhicit ← sGhicit + s[o]*x[o]
      vGhicit ← vGhicit + v[o]*x[o]
   if ((sGhicit ≤ M) and (vGhicit ≥ K))
      then success
      else failure
end
```

Este uşor de văzut că algoritmul are complexitatea $O(n)$. Aşadar, puterea de a ghici reduce drastic complexitatea. Acest aspect va fi discutat mai pe larg în secţiunea 13.3.

## 13.2 Clasele $\mathbb{P}$ şi $\mathbb{NP}$

Notăm cu $\mathbb{P}$ clasa problemelor $P$ pentru care există un algoritm determinist care rezolvă $P$ în timp polinomial şi cu $\mathbb{NP}$ clasa problemelor $P$ pentru care există un algoritm nedeterminist care rezolvă $P$ în timp polinomial. Evident, are loc:

$$\mathbb{P} \subseteq \mathbb{NP}$$

Există foarte multe motive pentru a crede că incluziunea $\mathbb{P} \subseteq \mathbb{NP}$ este strictă, *i.e.*, $\mathbb{P} \subset \mathbb{NP}$. Algoritmii nedeterminişti constituie un instrument mult mai puternic decât cei determinişti şi până acum nu s-a putut găsi o metodă prin care algoritmi nedeterminişti să poată fi convertiţi în algoritmi determinişti în timp polinomial. Cel mai puternic rezultat care s-a putut dovedi este următorul:

**Teorema 13.1.** *Dacă o problemă $P$ aparţine clasei $\mathbb{NP}$, atunci există polinoamele $p(n)$ şi $q(n)$, astfel încât $P$ poate fi rezolvată de un algoritm determinist în timpul $O(p(n)2^{q(n)})$.*

*Demonstraţie.* Fără să restrângem generalitatea, presupunem că funcţia `choice` alege aleator din mulţimea $\{0,1\}$ şi că algoritmul nedeterminist „ghiceşte"structura prin $q(n)$ apeluri ale lui `choice`. Mai presupunem că verificarea structurii se face în timpul $O(p(n))$. Algoritmul determinist va genera şi verifica toate cele $2^{q(n)}$ structuri. Evident, această explorare necesită timpul $O(p(n)2^{q(n)})$. sfdem

Pentru a arăta că o anumită problemă este într-o clasă dată, este necesară găsirea unui algoritm care să rezolve problema şi să îndeplinească cerinţele din definiţia clasei. Dovedirea neapartenenţei este mai dificilă şi se face pe o cale indirectă. O asemenea metodă de demonstrare este reducerea.

**Definiţia 13.1.** *Fie $P$ şi $Q$ două probleme şi $g(n)$ un polinom. Spunem că $P$ este transformată în timpul $O(g(n))$ în problema $Q$ dacă există o funcţie $t$ astfel încât:*

*1. $t$ are complexitatea timp $O(g(n))$;*

*2. $t$ transformă o instanţă $p \in P$ într-o instanţă $t(p) \in Q$;*

*3. pentru orice instanţă $p \in P$, $p$ şi $t(p)$ au acelaşi răspuns (valoare de adevăr).*

*Notăm $P \propto_{g(n)} Q$ şi citim $P$ se reduce polinomial la $Q$. Dacă precizarea polinomului $g(n)$ nu interesează, atunci notăm numai $P \propto Q$.*

Are loc următorul rezultat:

**Teorema 13.2.** *a) Dacă $P$ are complexitatea timp $\Omega(f(n))$ şi $P \propto_{g(n)} Q$, atunci $Q$ are complexitatea timp $\Omega(f(n) - g(n))$.*

*b) Dacă $Q$ are complexitatea $O(f(n))$ şi $P \propto_{g(n)} Q$, atunci $P$ are complexitatea $O(f(n) + g(n))$.*

*Demonstraţie.* Fie $B$ un algoritm care rezolvă $Q$. Următorul algoritm, notat cu $A$, rezolvă $P$:

1. calculează $t(p)$;
2. apelează $B$ pentru intrarea $t(p)$.

a) Deoarece $P$ are complexitatea $\Omega(f(n))$, rezultă că există $c > 0$ astfel încât $T_A(n) \geq c \cdot f(n)$. Dar:

$$T_A(n) = c \cdot g(n) + T_B(n)$$

de unde:

$$T_B(n) \geq c \cdot f(n) - c' \cdot g(n)$$

care arată că:

$$T_B(n) = \Omega(f(n) - g(n))$$

b) Deoarece $Q$ are complexitatea $O(f(n))$, rezultă $T_B \leq c \cdot f(n)$. Avem:

$$T_A(n) \leq c \cdot f(n) + c' \cdot g(n) = O(f(n) + g(n)).$$

Toate inegalităţile de mai sus au loc pentru $n \geq n_0$, unde $n_0$ este o constantă convenabil aleasă. sfdem

Iată cum poate fi folosită reducerea la demonstrarea neapartenenţei la $\mathbb{P}$: dacă $P \notin \mathbb{P}$ şi $P \propto Q$, atunci $Q \notin \mathbb{P}$. De asemenea, reducerea poate fi utilizată şi pentru a dovedi apartenenţa: dacă $Q \in \mathbb{P}$ şi $P \propto Q$, atunci $P \in \mathbb{P}$.

Deci pentru a putea a arăta că există incluziunea strictă $\mathbb{P} \subset \mathbb{NP}$, este necesar să găsim o problemă care este în $\mathbb{NP}$ şi nu este în $\mathbb{P}$. Pentru acest rol ar putea candida acele probleme $P$ cu proprietatea că pentru orice altă problemă $Q \in \mathbb{NP}$ are loc $Q \propto P$. Această observaţie justifică următoarea definiţie.

**Definiţia 13.2.** *a) Problema $P$ este* $\mathbb{NP}$*-dificilă dacă $Q \propto P$ pentru orice $Q \in \mathbb{NP}$.*

*b) Problema $P$ este* $\mathbb{NP}$*-completă dacă $P \in \mathbb{NP}$ şi $P$ este* $\mathbb{NP}$*-dificilă.*

Prima problemă care a fost dovedită a fi $\mathbb{NP}$-completă este cunoscută sub numele de SATISFIABILITATE (pe scurt *SAT*).

**Definiţia 13.3.** Problema SAT
*Fie $X = \{x_0, \ldots, x_{n-1}\}$ o mulţime de variabile. Un* literal *este o variabilă $x$ sau negaţia sa $\overline{x}$. O atribuire este o funcţie $\alpha : X \to \{0,1\}$. Atribuirea $\alpha$ se extinde la literale astfel:*

$$\alpha(u) = \begin{cases} \alpha(x) & \text{dacă } u = x \in X, \\ \neg\alpha(x) & \text{dacă } u = \overline{x}, x \in X \end{cases}$$

*O* clauză *este o mulţime finită $c$ de literale. Clauza $c$ este satisfăcută de atribuirea $\alpha$ dacă $\alpha(u) = 1$ pentru cel puţin un $u \in c$. Problema satisfiabilităţii constă în a determina dacă, pentru o secvenţă de clauze $C = (c_0, \ldots, c_{q-1})$, există o atribuire $\alpha$ care satisface orice clauză care apare în $C$.*

**Observaţie.** SAT poate fi reformulată astfel: dată o formulă din calculul propoziţional în formă normală conjunctivă, să se decidă dacă există o atribuire pentru variabile pentru care formula este adevărată. În alte lucrări, forma normală conjunctivă este înlocuită cu forma normală disjunctivă. Evident, cele două probleme sunt echivalente: există algoritmi polinomiali care transformă o formă normală în cealaltă.

sfobs

Următoarea teoremă i se datorează lui Steven Cook (1971).

**Teorema 13.3.** *SAT este NP-completă.*

*Demonstraţie.* Mai întâi dovedim că $SAT \in \mathbb{NP}$. Un algoritm nedeterminist care rezolvă SAT este construit schematic după cum urmează:

1. ghiceşte o atribuire $\alpha$;
2. evaluează clauzele $c_i$, $i = 0, \ldots, q-1$;
3. dacă $\alpha$ satisface orice clauză $c_i$, $i = 0, \ldots, q-1$, atunci algoritmul se opreşte cu succes;
4. altfel, se opreşte cu eşec.

Acum vom arăta că pentru orice problemă $P \in \mathbb{NP}$ avem $P \propto SAT$. Fie $P \in \mathbb{NP}$. Există un algoritm nedeterminist $A$ care rezolvă în timp polinomial $P$. Fără să restrângem generalitatea, facem asupra lui $A$ următoarele presupuneri:

- lucrează numai cu numere naturale;
- o locaţie de memorie poate memora numere oricât de mari; un număr $x$ este reprezentat pe $O(\log x)$ biţi (cifre binare 0 şi 1);
- instrucţiunile sunt etichetate cu $0, 1, 2, \ldots$ (precizăm că $A$ are un număr finit de instrucţiuni);
- pentru orice intrare de mărime $n$:
  - utilizează $p(n)$ locaţii, unde $p(n)$ este un polinom;
  - notăm aceste locaţii cu $1, 2, \ldots, p(n)-1$;
  - există un polinom $q(n)$ cu proprietatea că mărimea reprezentărilor numerelor memorate în locaţii nu depăşeşte $q(n)$.

Problema $P$ poate fi reformulată astfel: dată o intrare $x$ de lungime $n$ pentru $A$, să se decidă dacă există un calcul pentru care instrucţiunea din configuraţia finală este success. Vom descrie schematic un algoritm care transformă în timp polinomial mulţimea calculelor posibile pentru o intrare dată într-o formulă logică din calculul propoziţional. Formulele vor fi construite cu ajutorul următoarelor variabile booleene:

1. $B_{i,j,t}$ reprezintă valoarea bitului $j$ din reprezentarea numărului memorat în locaţia $i$ la momentul $t$ ($0 \leq i \leq p(n)-1, 0 \leq j \leq q(n)-1, 0 \leq t \leq T_A(n)$);
2. $S_{k,t}$ arată dacă instrucţiunea de etichetă $k$ este cea care este executată la momentul $t$.

Formula logică ataşată unei intrări $x$, este de forma $F(A,x) = F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge F_6$ unde:

$F_1$ reprezintă starea iniţială;
$F_2$ reprezintă faptul că instrucţiunea de etichetă 1 este prima instrucţiune care se execută;
$F_3$ reprezintă faptul că după $t$ paşi există exact o instrucţiune care se poate executa, $t = 0, 1, \ldots, T_A(n)-1$;
$F_4$ reprezintă calculul de etichete pentru instrucţiuni;
$F_5$ reprezintă schimbările stărilor memoriei;
$F_6$ spune dacă la momentul $T_A(n)$ s-a executat instrucţiunea `success` sau nu.

Nu vom da modurile de construire a fiecărei subformule pentru cazul general, dar vom considera un exemplu. Presupunem pentru moment că $A$ este următorul algoritm:

```
procedure A(x)
  1: if (x > 2)
  2: then y ← x*x
  3: else y ← x*x*x
  4: success
end
```

Notăm cu 0, 1, 2 locaţiile de memorie care memorează pe 2, $x$ şi respectiv $y$. Presupunem că toate numerele sunt reprezentate binar. Rezultă: $n = \log x$, $p(n) = 3$ (polinom constant), $q(n) = 3n$ şi $T_A(n) = 3$[1]. Pentru a face lucrurile şi mai precise, presupunem $x = 3$ care implică $n = 2$. Subformulele de mai sus sunt calculate după cum urmează:

1. Iniţial, locaţia 0 memorează 10 (reprezentarea lui 2), iar locaţia 1 pe 11 (reprezentarea lui $x$):

$$F_1 = B_{0,0,0} \wedge \overline{B}_{0,1,0} \wedge B_{1,0,0} \wedge B_{1,1,0}$$

2. $F_2 = S_{1,0} \wedge \overline{S}_{2,0} \wedge \overline{S}_{3,0} \wedge \overline{S}_{4,0}$.

3. $F_3 = G_0 \wedge G_1 \wedge G_2$, unde $G_t$ spune că la momentul $t$ se execută exact o instrucţiune. Putem lua:

$$G_t = G_{1,t} \oplus G_{2,t} \oplus G_{3,t} \oplus G_{4,t}$$

unde $G_{k,t}$ spune că la momentul $t$ se execută instrucţiunea $k$. De exemplu $G_{2,t} = \overline{S}_{1,t} \wedge S_{2,t} \wedge \overline{S}_{3,t} \wedge \overline{S}_{4,t}$.

[1] Calculul complexităţii timp este artificial, deoarece, în general, timpul necesar efectuării unei înmulţiri depinde de mărimea operanzilor. Pentru a simplifica prezentarea, am presupus că orice instrucţiune se realizează într-o unitate de timp.

4. $F_4 = H_0 \wedge H_1 \wedge H_2$, unde $H_t$ spune cum se calculează adresa de instrucţiune la momentul $t$. Putem considera:

$$H_t = H_{1,t} \wedge H_{2,t} \wedge H_{3,t} \wedge H_{4,t}$$

cu $H_{k,t}$ reprezentând modul în care se calculează adresa dacă la momentul $t$ se execută instrucţiunea de adresă $k$. De exemplu, $H_{2,t} = \overline{S}_{2,t} \vee S_{4,t-1}$.

5. $F_5 = K_0 \wedge K_1 \wedge K_2$, unde $K_t$ arată cum se schimbă memoria la momentul $t$. Schimbarea memoriei depinde de instrucţiunea care se execută la momentul $t$:

$$K_t = K_{1,t} \wedge K_{2,t} \wedge K_{3,t} \wedge K_{4,t}$$

unde $K_{k,t}$ reprezintă modul în care se schimbă memoria dacă la momentul $t$ se execută instrucţiunea $k$. De exemplu, faptul că instrucţiunea 1 lasă memoria neschimbată se exprimă prin formula:

$$K_{1,t} = \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{j=5} ((B_{i,j,t} \wedge B_{i,j,t+1}) \vee (\overline{B}_{i,j,t} \wedge \overline{B}_{i,j,t+1}))$$

unde $2 = p(n) - 1$, $5 = q(n) - 1$.

6. $F_6 = S_{4,3}$.

Cititorul este invitat să verifice că toate cele şase subformule pot fi determinate pentru cazul general.

Pentru o intrare $x$ de lungime $n$, există un calcul al lui $A$ care se termină cu succes numai dacă există o atribuire de valori booleene pentru variabilele $B(i,j,t)$ şi $S(k,t)$ care satisface $F(A,x)$. Transformarea formulei $F(A,x)$ în formă normală conjunctivă se face cu unul dintre algoritmii cunoscuţi. sfdem

Utilizând această teoremă şi tehnica reducerii, s-a putut dovedi că multe probleme sunt NP-complete. Vom mai discuta despre aceste probleme în capitolul 13.

## 13.3 Probleme NP-complete

Reamintim că o problemă $P$ este NP-completă dacă:

- este în NP, *i.e.*, există un algoritm nedeterminist care rezolvă $P$ în timp polinomial (echivalent, există un algoritm determinist care rezolvă $P$ în timp exponenţial);
- este NP-dificilă, *i.e.*, orice altă problemă $Q$ din NP se reduce polinomial la $P$.

Avem $\mathbb{P} = \mathbb{NP}$ dacă există un algoritm determinist care rezolvă în timp polinomial o problemă NP-completă. Până astăzi nu a fost găsit un asemenea algoritm. De fapt, toţi cercetătorii din informatica teoretică consideră că egalitatea nu are loc. *i.e.*, există incluziunea strictă $\mathbb{P} \subset \mathbb{NP}$ şi, de aceea, pare o nebunie orice încercare de găsire a unui asemenea algoritm.

Din păcate, multe probleme practice s-au dovedit a fi NP-complete. Există probleme NP-complete în calculul numeric, în geometrie, în algebră, în procesarea

grafurilor, în procesarea şirurilor etc. De aceea, un bun proiectant de algoritmi trebuie să înţeleagă bine elementele de bază ale NP-completitudinii. Aceste elemente includ formalizarea abstractă a unei probleme, instrumente cu care se poate dovedi că o anumită problemă este NP-completă şi ce trebuie făcut după ce s-a dovedit că o problemă este NP-completă.

### 13.3.1 Cum se poate dovedi NP-completitudinea

În [GJ79] sunt propuse următoarele tehnici prin care se poate dovedi că o problemă $P$ este NP-completă:

**Reducerea.** Se arată că $P \in \mathbb{NP}$ prin proiectarea unui algoritm nedeterminist polinomial care rezolvă $P$. Pentru a arăta că $P$ este NP-dificilă se utilizează reducerea. Se ştie că problema $Q$ este NP-completă (deci NP-dificilă) şi se arată că $Q \propto P$. Metoda poate fi reprezentată schematic prin:

$$(Q \text{ NP-completă},\ Q \propto P) \Rightarrow P \text{ NP-completă}$$

Considerăm ca exemplu problema 3SAT care se obţine din SAT impunând restricţia ca fiecare clauză să fie formată exact din trei literale. Faptul că 3SAT $\in$ NP rezultă din observaţia că orice algoritm care rezolvă SAT rezolvă de asemenea 3SAT. Vom arăta că SAT $\propto$ 3SAT descriind modul în care se construieşte o instanţă a problemei 3SAT plecând de la o instanţă a problemei SAT. Prezentăm, pentru câteva cazuri particulare, cum o clauză oarecare $c$ poate fi scrisă cu ajutorul clauzelor cu exact trei literale.

1. $c = u_1$. Considerăm $c' = (u_1 \vee y_1 \vee y_2) \wedge (u_1 \vee y_1 \vee \overline{y}_2) \wedge (u_1 \vee \overline{y}_1 \vee y_2) \wedge (u_1 \vee \overline{y}_1 \vee \overline{y}_2)$, unde $y_1$ şi $y_2$ sunt două variabile noi.
2. $c = u_1 \vee u_2$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_1 \vee u_2 \vee \overline{y}_1)$, unde $y_1$ este o variabilă nouă.
3. $c = u_1 \vee u_2 \vee u_3 \vee u_4$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_3 \vee u_4 \vee \overline{y}_1)$, unde $y_1$ este o variabilă nouă.
4. $c = u_1 \vee u_2 \vee u_3 \vee u_4 \vee u_5$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_3 \vee \overline{y}_1 \vee y_2) \wedge (u_4 \vee u_5 \vee \overline{y}_2)$, unde $y_1$ şi $y_2$ sunt variabile noi.

În toate cazurile, $c$ este satisfiabilă dacă şi numai dacă $c'$ este satisfiabilă. Extensia se referă la atribuirea de valori pentru variabilele noi $y_i$. Cititorul este invitat să găsească singur regula generală de construcţie a clauzei $c'$ şi să arate că această construcţie este făcută în timp polinomial.

Un alt exemplu îl constituie problema $V$-acoperirii într-un graf. O $V$-acoperire în graful $G = \langle V, E \rangle$ este o submulţime de vârfuri $V' \subseteq V$ cu proprietatea că fiecare muchie din $E$ are o extremitate în $V'$, *i.e.*, $(\forall \{i, j\} \in E) i \in V' \vee j \in V'$. Problema $V$-acoperirii este:

**$V$-acoperire (VA)**
*Instanţă.* Un graf $G = \langle V, E \rangle$ şi $k \in \mathbb{Z}_+$.
*Întrebare.* Există o $V$-acoperire $V'$ cu $\#V' \leq k$?

Se poate arăta că 3SAT $\propto$ VA. De exemplu, graful $G$ construit pentru $C = (x_1 \vee \overline{x}_3 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_4) = c_1 \wedge c_2$ este reprezentat în figura 13.1. Considerăm $k = 8 = n + 2m$. Orice submulţime $V' \subseteq V$ defineşte o atribuire $\alpha_{V'}$ dată prin $\alpha_{V'}(x_i) = true \iff x_i \in V'$ şi $\alpha_{V'}(x_j) = false \iff \overline{x}_j \in V'$. Dacă $V'$ este o $V$-acoperire, rezultă $\{i, j\} \in E$ dacă şi numai dacă $i \in V' \vee j \in V'$. Dacă $\#V' \leq 8$, atunci orice clauză $c_i$ are numai două vârfuri în $V'$. Deci există un al treilea vârf care nu este în $V'$. Dar din acest vârf există o muchie la un $x_j$ sau un $\overline{x}_j$ care apare în $c_i$ şi deci $\alpha_{V'}(x_j) = true$ sau $\alpha_{V'}(\overline{x}_j) = true$. În final, obţinem că orice $V$-acoperire $V'$ cu $\#V' \leq 8$ defineşte o atribuire care satisface $C$.

**Exerciţiul 13.1.** Să se arate că $VA$ este în NP.

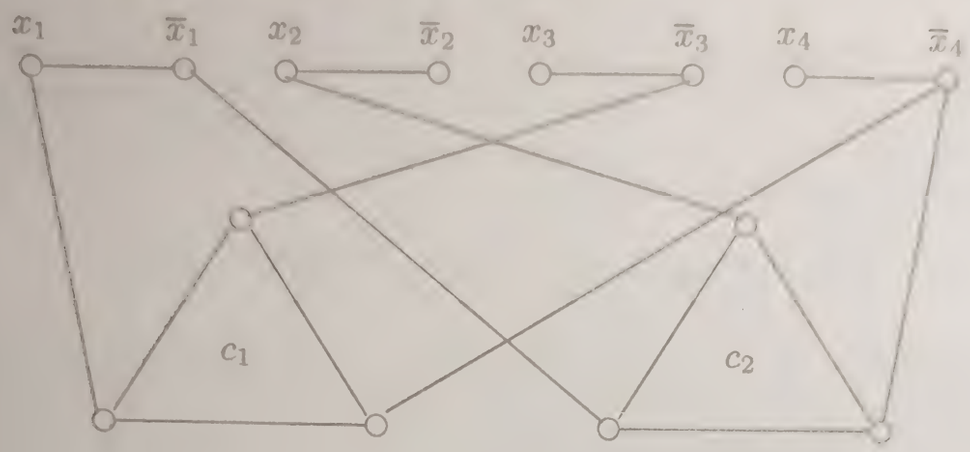



Figura 13.1: Graf asociat unei mulţimi de clauze

**Restricţia.** Se cunoaşte că problema $Q$ este NP-completă. Se arată că $P$ este NP-completă dovedind că $Q$ este un caz special al problemei $P$, *i.e.*, orice instanţă a lui $Q$ se obţine dintr-o instanţă a lui $P$ prin adăugarea de noi restricţii. Metoda poate fi reprezentată schematic prin:

($Q$ NP-completă, $Q$ caz special al lui $P$) $\Rightarrow$ $P$ NP-completă

Considerăm următoarele două probleme:

**Circuit Hamiltonian într-un digraf (CHD)**
*Instanţă.* Un digraf $D = \langle V, A \rangle$.
*Întrebare.* Conţine $D$ un circuit hamiltonian (circuit care trece prin toate vârfurile din $V$)?

**Circuit Hamiltonian într-un graf (CHG)**
*Instanţă.* Un graf $G = \langle V, E \rangle$.
*Întrebare.* Conţine $G$ un circuit hamiltonian?

Se ştie că CHG este NP-completă (se poate arăta, de exemplu, că VA $\propto$ CHG [GJ79]). Restricţionând CHD la acele digrafuri $D = \langle V, A \rangle$ cu proprietatea că pentru orice două vârfuri $i \neq j$ există perechea de arce $\{i, j\}, \{j, i\} \in A$, obţinem o problemă echivalentă cu CHG. De aici, rezultă că CHD este NP-completă.

**Înlocuirea locală.** Se bazează pe reducere. Se alege o problemă NP-completă şi se pun în evidenţă câteva aspecte ale instanţelor. Acestea formează o colecţie de unităţi de bază. Înlocuind fiecare unitate de bază cu o structură echivalentă, se obţine o instanţă a problemei $P$ (despre care dorim să arătăm că este NP-completă). Un exemplu de aplicare a acestei tehnici îl constituie reducerea SAT $\propto$ 3SAT.

**Proiectarea componentelor.** Şi aceasta se bazează pe reducere. Constituenţii unei instanţe a problemei $P$ sunt utilizaţi pentru a proiecta anumite „componente"care apoi sunt combinate pentru a crea o instanţă a unei probleme NP-complete cunoscute. Reducerea 3SAT $\propto$ VA este un exemplu de aplicare a acestei tehnici. Constituenţii unei instanţe 3SAT – clauzele şi atribuirile care satisfac clauzele – au fost utilizate pentru proiectarea componentelor – graf şi alegerea unei $V$-acoperiri.

### 13.3.2 Despre euristici

Ce trebuie să facem atunci când avem de rezolvat o problemă practică despre care am dovedit că este NP-completă? Din păcate nu putem da răspunsuri care să se constituie în reţete „sigure". Pentru problemele de optimizare, *algoritmii de aproximare* (euristicile) constituie o soluţie aplicată cu succes. Algoritmii de aproximare se bazează pe ideea „mai bine o soluţie aproape optimă obţinută în timp polinomial (deci util) decât soluţia exactă obţinută într-un timp foarte mare, deci inefectiv". Cei mai mulţi algoritmi de aproximare se bazează pe metoda greedy. Considerăm ca exemplu *problema comis-voiajorului*:

**Comis-voiajor (CV)**
*Instanţă.* Un graf ponderat $\langle\langle V, E\rangle, c\rangle$ cu $c(i,j) \in \mathbb{Z}_+$ pentru orice muchie $\{i,j\} \in E$ şi $K \in \mathbb{Z}_+$.
*Întrebare.* Există un circuit hamiltonian de cost $\leq K$?

**Exerciţiul 13.2.** Să se arate că CV este NP-completă.

Cititorul a observat deja că problema de optimizare corespunzătoare lui CV constă în determinarea unui circuit hamiltonian de cost minim. Numele problemei vine de la următoarea modelare: un comis-voiajor trebuie să treacă prin $n$ oraşe (vârfurile grafului). O muchie în graf corespunde existenţei unui drum între oraşele din extremităţi, iar costul muchiei reprezintă lungimea drumului. Problema constă în determinarea unui tur al celor $n$ oraşe de lungime minimă (pentru ca efortul comis-voiajorului să fie minim). În cazul în care are loc o relaţie de tip „inegalitatea triunghiului":

$$c(i,j) + c(j,k) \geq c(i,k)$$

se poate da o schemă de aproximare pentru CV bazată pe construcţia unui arbore de cost minim (prin algoritmul lui Kruskal sau al lui Prim, exerciţiul 9.9) şi parcurgerea în preordine a acestuia.

Atunci când problemele au dimensiuni rezonabile, se poate aplica metoda programării dinamice sau backtracking. De exemplu, pentru problema rucsacului, varianta discretă, algoritmul backtracking are o eficienţă satisfăcătoare. Dacă valoarea

$M$, care dă capacitatea rucsacului este mică, atunci se poate aplica cu succes şi algoritmul construit prin metoda programării dinamice. De fapt, dacă $M$ este reprezentat unar, atunci algoritmul are complexitate polinomială. Complexitatea timp este un polinom în lungimea reprezentării unare a datelor de intrare. Un asemenea algoritm este numit *algoritm cu complexitate pseudopolinomială*.

### 13.3.3 Câteva probleme NP-complete importante

În această secţiune prezentăm o listă cu câteva probleme NP-complete. Câteva dintre ele au fost studiate deja în capitolele anterioare. Pentru o listă mai extinsă se poate consulta [GJ79].

#### 13.3.3.1 Mulţimi

**Submulţimea de sumă dată**

*Instanţă.* O mulţime $A$, o mărime $s(a) \in \mathbb{Z}_+$, pentru orice $a \in A$ şi $K \in \mathbb{Z}_+$.

*Întrebare.* Există o submulţime $A' \subseteq A$ pentru care suma mărimilor elementelor din $A'$ să fie exact $K$?

**Submulţimea de produs dată**

*Instanţă.* O mulţime $A$, o mărime $s(a) \in \mathbb{Z}_+$, pentru orice $a \in A$ şi $K \in \mathbb{Z}_+$.

*Întrebare.* Există o submulţime $A' \subseteq A$ pentru care produsul mărimilor elementelor din $A'$ să fie exact $K$?

#### 13.3.3.2 Planificare

**Planificarea procesoarelor**

*Instanţă.* O mulţime $P$ de programe, un număr $m$ de procesoare, un timp de execuţie $t(p)$, pentru fiecare $p \in P$, şi un termen $D$.

*Întrebare.* Există o planificare a procesoarelor pentru $P$ astfel încât orice program să fie executat înainte de termenul $D$?

#### 13.3.3.3 Logică

**Satisfiabilitate (SAT)**

*Instanţă.* O mulţime $X$ de variabile şi o mulţime $C$ de clauze peste $X$.

*Întrebare.* Există o atribuire $\alpha : X \to Boolean$ care să satisfacă $C$?

**3-Satisfiabilitate (3SAT)**

*Instanţă.* O mulţime $X$ de variabile şi o mulţime $C$ de clauze peste $X$ astfel încât orice clauză $c \in C$ are $|c| = 3$.

*Întrebare.* Există o atribuire $\alpha : X \to Boolean$ care să satisfacă $C$?

#### 13.3.3.4 Algebră

**Congruenţe pătratice**
*Instanţă.* Întregii pozitivi $a, b, c$.
*Întrebare.* Există un întreg pozitiv $x < c$ astfel încât $x^2 \equiv a \pmod{b}$?

**Ecuaţii diofantice pătratice**
*Instanţă.* Întregii pozitivi $a, b, c$.
*Întrebare.* Există $x, y \in \mathbb{Z}_+$ astfel încât $ax^2 + by = c$?

#### 13.3.3.5 Programare matematică

**Programare întreagă**
*Instanţă.* O mulţime $X$ de perechi $(x, b)$, unde $x = (x_1, \ldots, x_m), x_i, b \in \mathbb{Z}$, o secvenţă $c = (c_1, \ldots, c_m)$ şi un întreg $K$.
*Întrebare.* Există $y = (y_1, \ldots, y_m)$ astfel încât:
1. $\sum_i x_i y_i \leq b$ pentru orice $(x, b) \in X$;
2. $\sum_i c_i y_i \geq K$?

**Rucsac 0/1**
*Instanţă.* O mulţime $O$ (obiectele), o mărime $w(o) \in \mathbb{Z}_+$ şi o valoare $v(o) \in \mathbb{Z}_+$ pentru fiecare obiecte $o \in O$, o restricţie $M \in \mathbb{Z}_+$ şi un scop $K \in \mathbb{Z}_+$.
*Întrebare.* Există o submulţime $O' \subseteq O$ astfel încât

$$\sum_{o \in O'} w(o) \leq M \text{ şi } \sum_{o \in O'} v(o) \geq K?$$

#### 13.3.3.6 Grafuri

**$K$-Colorare**
*Instanţă.* Un graf $G = \langle V, E \rangle$ şi $k \in \mathbb{Z}_+$.
*Întrebare.* Există o colorare cu $k$ culori a grafului $G$?

**$V$-acoperire (VA)**
*Instanţă.* Un graf $G = \langle V, E \rangle$ şi $k \in \mathbb{Z}_+$.
*Întrebare.* Există o $V$-acoperire $V'$ cu $\#V' \leq k$?

**Circuit Hamiltonian într-un digraf (CHD)**
*Instanţă.* Un digraf $D = \langle V, A \rangle$.
*Întrebare.* Conţine $D$ un circuit hamiltonian (circuit care trece prin toate vârfurile din $V$)?

**Circuit Hamiltonian într-un graf (CHG)**
*Instanţă.* Un graf $G = \langle V, E \rangle$.
*Întrebare.* Conţine $G$ un circuit hamiltonian?

**Comis-voiajor (CV)**
*Instanţă.* Un graf ponderat $\langle \langle V, E \rangle, c \rangle$ cu $c(i, j) \in \mathbb{Z}_+$ pentru orice muchie $\{i, j\} \in E$ şi $K \in \mathbb{Z}_+$.
*Întrebare.* Există un circuit hamiltonian de cost $\leq K$?

**Cel mai lung drum**
*Instanţă.* Un graf ponderat $G = \langle V, E \rangle$ cu ponderele $\ell : E \to \mathbb{Z}_+$, un întreg pozitiv $K$ şi două vârfuri distincte $i$ şi $j$.
*Întrebare.* Există în $G$ un drum de la $i$ la $j$ de lungime $\geq K$?

## 13.4 Exerciţii

**Exerciţiul 13.3.** Să se proiecteze un algoritm cu complexitatea timp pseudopolinomială care rezolvă problema determinării unei submulţimi de sumă dată.

**Exerciţiul 13.4.** [CLR93] Se consideră următorul algoritm greedy ca algoritm de aproximare pentru $V$-acoperire: pasul de alegere greedy selectează vârful cu cel mai mare grad şi elimină toate muchiile incidente în el. Pentru o instanţă dată, notăm cu $C$ soluţia dată de algoritmul de aproximare şi cu $C^*$ soluţia optimă.

1. Să se găsească un exemplu pentru care $C \neq C^*$.
2. Să se găsească un exemplu pentru care $\frac{\#C}{\#C^*} > 2$.

**Exerciţiul 13.5.** Să se proiecteze un algoritm eficient care determină o $V$-acoperire optimă pentru un arbore în timp liniar.

## 13.5 Referinţe bibliografice

Pentru o tratare mai detaliată a subiectului recomandăm [GJ79, Mel84b, LG86, Koz92, AHU74, HS84, Giu04]. O trecere în revistă a algoritmilor de aproximare, în special pentru probleme de combinatorică şi de programare liniară, poate fi găsită în [Vaz04]. Pentru o abordare sistematică a problemelor dificile, incluzând algoritmi aleatorii, poate fi consultată [Hro04]. O paletă mai largă de euristici se găseşte în [MF04].

# Bibliografie


[AHU74] A.V. Aho, J.E. Hopcroft şi J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[AA78] S. Alagic şi M.A. Arbib. *The Design of Well-Structured and Correct Programs.* Springer-Verlag, 1978.

[AS84] S. Arikawa şi S.Shiraishi. „Pattern Matching Machines for Replacing Several Character Strings", *Bulletin of Informatics and Cybernetics*, volumul 21, pp. 101-112, martie 1984.

[Baa78] Sarah Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, Reading, Massachusetts, 1978.

[BD62] R.E. Bellman şi S.E. Dreyfus. *Applied Dynamic Programming.* Princeton University Press, 1962.

[BG00] S. Baase şi A. Van Gelder. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, Reading, Massachusetts, 2000.

[CB02] M. Craus şi C. Bârsan. *Structuri de date şi algoritmi.* Editura „Gh. Asachi", Iaşi, 2002.

[CLR93] T.H. Cormen, C.E. Leiserson şi R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1993.

[CLR00] T.H. Cormen, C.E. Leiserson şi R.L. Rivest. *Introducere în algoritmi.* Computer Libris Agora, 2000.

[Coh90] E. Cohen. *Programming in the 1990s: An Introduction to the Calculation of the Programs.* Springer–Verlag, 1990.

[Cro92] Cornelius Croitoru. *Tehnici de bază în optimizarea combinatorie.* Editura Universităţii „Al.I. Cuza", Iaşi, 1992.

[Cro02] Cornelius Croitoru. *Introducere în proiectarea algoritmilor paraleli.* Editura Matrix Rom, Bucureşti, 2002.

[Giu04] C.A. Giumale. Introducere în ANALIZA ALGORITMILOR. Teorie şi aplicaţie. Polirom, 2004.

[GJ79] M.R. Garey şi D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H.Freeman and Company, San Francisco, 1979.

[Gri81] D. Gries. *The Science of Programming.* Springer–Verlag, 1981.

[HE87] G.H. Hale şi R.J. Easton. *Applied Data structures Using Pascal.* D.C.Heath and Company, 1987.

[Hei96] G. Heileman. *Data Structures, Algorithms and Object-Oriented Programming.* McGraw-Hill, 1996.

[Hro04] J. Hromkovic. *Algorithmics for Hard Problems.* Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.

[HS84] E. Horowitz şi S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1984.

[HSAF93] E. Horowitz, S. Sahni şi S. Anderson-Freed. *Fundamentals of Data Structures in C.* Computer Science Press, 1993.

[Kao08] M.-Y. Kao. *Encyclopedia of Algorithms.* Springer, 2008.

[KGGK94] V. Kumar, A. Grama, A. Gupta şi G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin Cummings, 1994.

[Knu74] D.E. Knuth. *Algoritmi fundamentali*, volumul 1: *Tratat de programarea calculatoarelor.* Editura Tehnică, Bucureşti, 1974.

[Knu76] D.E. Knuth. *Sortare şi căutare*, volumul 3: *Tratat de programarea calculatoarelor.* Editura Tehnică, Bucureşti, 1976.

[Koz92] D.C. Kozen. *The Design and Analysis of Algorithms.* Springer Verlag, 1992.

[Kro79] L.I. Kronsjo. *Algorithms: Their Complexity and Efficiency.* John Wiley & Sons, 1979.

[LG86] L. Livovschi şi H. Georgescu. *Sinteza şi analiza algoritmilor.* Editura Ştiinţifică şi Enciclopedică, Bucureşti, 1986.

[Luc93] D. Lucanu. *Programarea algoritmilor: Tehnici elementare.* Editura Universităţii „Al.I. Cuza", Iaşi, 1993.

[Luc96] D. Lucanu. *Bazele proiectării programelor şi algoritmilor.* Curs multiplicat la Editura Universităţii „Al.I. Cuza", Iaşi, Romania, 1996, volumul 1: *Modele de calcul*, volumul 2: *Tehnici de programare*, volumul 3 *Proiectarea algoritmilor.*

[Man89] U. Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, Reading, 1989.

[MB00] R. Miller şi L. Boxer. *Algorithms Sequential & Parallel: A Unified Approach.* Prentice Hall, New Jersey, 2000.

[Mel79] K. Melhorn. Dynamic data structures. În J.W. de Bakker şi J. van Leeuwen (ed.), *Foundations of Computer Science III*, volumul 1. Mathematical Centre Tracts, Amsterdam, 1979.

[Mel84a] K. Melhorn. *Data Structures and Algorithms 1: Sorting and Searching* Springer Verlag, 1984.

[Mel84b] K. Melhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* Springer Verlag, 1984.

[MF04] Z. Michalewicz şi D.B. Fogel. *How to Solve It: Modern Heuristics.* Springer, 2004.

[MS91] B.M.E. Morret şi H.D. Shapiro. *Algorithms from P to NP: Design and Efficiency.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[Sah98] S. Sahni. *Data Structures and Algorithms in C++.* WCB McGraw-Hill, 1998.

[Sed88] R. Sedgewick. *Algorithms.* Adison-Wesley Publishing Company, 1988.

[Vaz04] V.V. Vazirani. *Approximation Algorithms.* Springer, 2004.

[Wei92] M.A. Weiss. *Data Structures and Algorithm Analysis in C*. Benjamin/Cummings Publishing Company, Inc., 1992.

[Wan80] M. Wand. *Induction, Recursion and Programming*. North Holland, 1980.

[Woo84] D. Wood. *Paradigms and Programming with Pascal*. Computer Science Press, 1984.

# Index


3SAT, 349, 352

algoritm, 13
  Strassen, 274
  complexitate pseudo-polinomială –, 352
  reducere polinomială, 345
  timpul de execuţie –, 31
  timpul mediu de execuţie –, 34
analizaă amortizată, 104
  - metoda agregării, 104
  - metoda conturilor, 106
  - metoda potenţialului, 106
arbore, 86
  - 2-3, 171
  - AVL, 161
  - B, 179
  - bicolor, 168
  - binar complet, 79
  - cu rădăcină, 86
  - de decizie
    - pentru căutare, 150
    - pentru sortare, 135
    lungimea externă a unui –, 151
    lungimea internă a unui –, 151
  - digital (trie), 190
  - echilibrat, 161
  - ordonat, 86
  - ponderat, 228
arbore binar, 71
  dimensiunea unui –, 71
  frontiera unui –, 71
  înălţimea unui –, 71
arbore binar de căutare
  - orientat pe frontieră, 158
  - orientat pe noduri interne, 157
arbore parţial
  - BFS, 98
  - DFS, 96
arbore ponderat pe frontieră,
    *vezi* arbore ponderat, 228
arce, 85
axioma de accesibilitate,
    *vezi* sistem accesibil, 246

backtracking, 309, 319
Bellman-Ford
  algoritmul –, 285, 304
Boyer-Moore, algoritmul –, 205
branch and bound
  - cu cost minim, 322
  - cu mărginire, 323
branch-and-bound, 309, 320
Bubble Sort,
    *vezi* sortare prin interschimbare, 124
Bucket Sort,
    *vezi* sortare prin distribuire, 141
buclă într-un graf, 83

căutare
  - în mulţimi total ordonate
    - Fibonacci, 150
    - binară, 150
    - divide et impera, 149
    - liniară, 150
    - prin interpolare, 154
    arbore de decizie, 150
  - aspect dinamic, 147
  - aspect static, 147
  - binară, 33
  - peste şiruri, 201
ciclu într-un graf, 84
circuit într-un graf, 84

clasă de arbori stabilă, 161
clauză, 346
    satisfacerea unei –, 346
coada, 62
coadă cu priorităţi, 79
coliziune,
        *vezi* dispersie, 185
configuraţie, 26
configuraţie finală, 26
configuraţie iniţială, 26
criteriu de alegere locală (*greedy*), 246

diametrul unui arbore, 274
digraf, 85
    – etajat, 280
    – etichetat, 85
    – ponderat, 85
    – tare conex, 85
    – tipul abstract –, 88
    parcurgerea sistematică a –, 94
Dijkstra
    algoritmul –, 250, 285
dispersie, 184
    – cu înlănţuire, 186
    – cu adresare deschisă, 188
distribuire prin segmentare,
        *vezi* sortare prin –, 141
divide-et-impera, 253
drum închis într-un graf, 84
drum într-un graf, 84
drum minim într-un digraf, 282

enumerare
    – a permutărilor
    – nerecursivă, 314
    – recursivă, 313
euristică, 351
execuţia programelor, 24
explorarea BFS,
        *vezi* parcurgerea sistematică a digrafurilor, 97
explorarea DFS,
        *vezi* parcurgerea sistematică a digrafurilor, 95
expresie regulată, 208

FFT,
        *vezi* transformata Fourier rapidă, 271
Fibonacci
    căutare –, 150
    funcţia lui –, 117
Floyd-Warshall
    algoritmul –, 285
Fourier,
        *vezi* transformata –, 270
funcţie
    – de dispersie, 185
funcţie obiectiv, 302

graf, 83
    – conex, 84
    – etichetat, 85
    – ponderat, 85
    – reprezentarea unui – ca digraf, 89
    – tipul abstract –, 86
graf general (pseudograf), 83
greedy, 225

hash,
        *vezi* dispersie, 184
Heap Sort
    vezi sortarea prin selecţie, 131

interpolare,
        *vezi* căutare în mulţimi total ordonate , 154

Knuth-Morris-Pratt, algoritmul –, 202
Kruskal, algoritmul lui –, 244

limbaj algoritmic, 13
lista circulară, 55
lista generalizată, 66
lista liniară, 41
lista liniraă, - ordonată51
liste de adiacenţă, 92
        dinamice, 92
        – tablouri, 98
literal, 346
lungime externă,
        *vezi* arbore de decizie, 151

lungime externă ponderată,
*vezi* arbore ponderat, 228
lungime internă,
*vezi* arbore de decizie, 151

mărimea unei instanţe, 30
matrice de adiacenţă, 89
matroid, 247
– ponderat, 247
max-*heap*, 79
maxim
– într-un tablou, 32
memorie, 14
Merge Sort,
*vezi* sortare prin interclasare, 255
mers într-un graf, 84
metoda bulelor,
*vezi* sortare prin interschimbare, 124
metoda par-impar,
*vezi* sortare prin interschimbare, 126
modelul arborilor de decizie pentru căutare, 150
modelul arborilor de decizie pentru sortare, 135
muchie (în graf), 83
muchie incidentă, 83
mulţime
– parţial ordonată,
*vezi* sortare topologică, 144
multigraf, 83

pădure, 101
Par-Impar Sort,
*vezi* sortare prin interschimbare, 126
paradigme de proiectare a algoritmilor, 217
parcurgerea în inordine nerecursivă, 109
Patricia, 196
pattern generalizat, 208
politică (optimă), 302
principiul de optim, 302
problemă
– NP-completă, 345, 348
– NP-dificilă, 345, 348
instanţă a unei –, 30
problemă, 28
– decidabilă, 29
– nedecidabilă, 29
– nerezolvabilă, 29
– parţial decidabilă, 29
– rezolvabilă, 29
– semidecidabilă, 29
problema
– celor $n$ regine, 327
– 3-satisfiabilităţii, 352
– K-colorării, 353
– V-acoperirii, 350, 353
– închisorii, 342
– înmulţirii optime a unui şir de matrici, 305
– înmulţirii polinoamelor, 274
– alocării optime a fişierelor, 248
– alocării resurselor, 279
– arborelui parţial de cost minim, 244
– celebrităţii, 218
– circuitului hamiltonian, 350, 353
– colorării grafurilor, 325
– comisului voiajor, 351, 353
– conectării tranzistorilor, 341
– congruenţei pătratice, 353
– determinării celui de-al $n$-lea număr Fibonacci, 274
– determinării minimului şi maximului simultan, 274
– distanţei între şiruri, 295
– dominoului,
*vezi* problema mozaicului, 341
– drumului de lungime maximă într-un graf, 353
– ecuaţiilor diofantice pătratice, 353
– evaluării polinoamelor, 274
– fortăreţei, 342
– galeriilor de artă, 341
– instructorului de schi, 242
– interclasării optime, 234
– labirintului, 340

- liniei orizontului, 264
- lungimii externe ponderate optime, 228
- mărcilor poştale, 340
- memorării programelor, 225
- memorării programelor II, 249
- mozaicului, 341
- mutării calului de şah, 341
- opririi, 29
- permutării maximale, 219
- planificării procesoarelor, 352
- procesării paralele, 341
- programării întregi, 353
- rucsacului, varianta continuă, 236
- rucsacului, varianta discretă, 285, 332
- satisfiabilităţii, 352
- schimbării banilor, 249, 340
- secvenţializării optime a activităţilor, 239
- selecţiei, 262
- submulţimii de produs dată, 352
- submulţimii de sumă dată, 330, 352
- subsecvenţei crescătoare maximale, 293
- triangularizării optime a unui poligon, 306
- turului bitonic minim, 306
- Rucsac 0/1, 343, 353

program, 14
programare dinamică, 278, 301
proprietatea de ereditate,
*vezi* matroid, 247
proprietatea de alegere locală, 227
proprietatea de interschimbare,
*vezi* matroid, 247
pseudograf, 83

Quick Sort,
*vezi* sortare rapidă, 258

Rabin-Karp, algoritmul –, 206
Radix Sort,
*vezi* sortare prin distribuire, 139
recursie, 113
- în cascadă, 116
- liniară, 113

reprezentarea expresiilor aritmetice, 78

SAT, 346, 352
secvenţă $h$-ordonată, 129
selecţie naivă,
*vezi* sortare prin selecţie, 131
selecţie sistematică,
*vezi* sortare prin selecţie, 131
selecţia sistematică, 131
sistem accesibil, 246
sortare
- a cuvintelor, 138
- cu micşorarea incrementului, 128
- externă, 123
- externă, 234
- internă, 123
- metoda inserţiei, 32
- prin distribuire, 138
- prin inserţie, 127
- prin interclasare, 255
- prin interschimbare, 124
- prin numărare, 142
- prin selecţie, 130
- rapidă, 258
- topologică, 144

sortare prin distribuire, 138
stabilitate a algoritmilor de sortare, 123
stiva, 57
structură
- dinamică înlănţuită, 22
- dinamică dublu înlănţuită, 24
- dinamică simplu înlănţuită, 24
- statică, 22

subgraf, 83
- indus, 83
- parţial, 83

substructură optimă, 278, 303
swap, 19

tabelă de dispersie, 185
tabelă de simboluri,
*vezi* dispersie, 184
tablou, 20
- 1-dimensional, 20

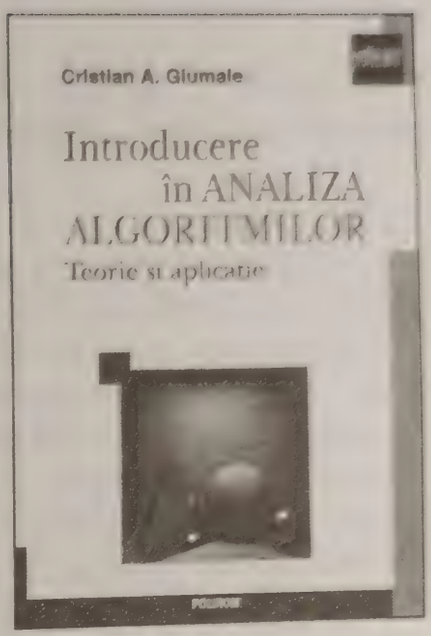
– 2-dimensional, 21
cu auto-organizare, 148
tehnici de derecursivare, 109
tip de date, 13
– abstract, 13
domeniul unui –, 13
transformata Fourier, 270
– rapidă, 271
transformata Fourier discretă, 270

union-find, structura –, 103

vârf (în graf), 83
vârfuri adiacente, 83
vârfuri pendante,
*vezi* arbore de decizie, 150
variabilă, 14
– dinamică, 15

Cristian A. Giumale

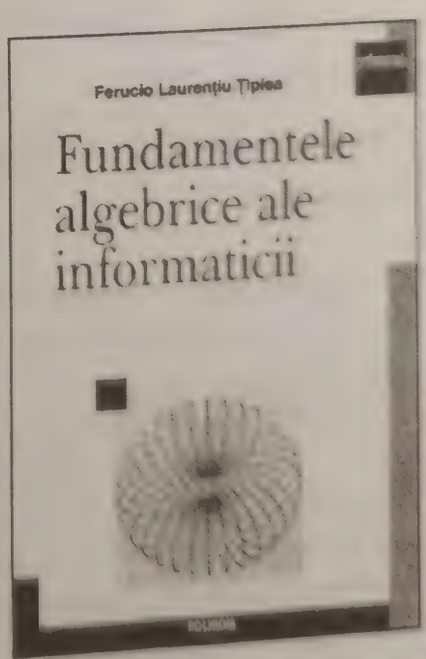
## Introducere în analiza algoritmilor. Teorie şi aplicaţie




Volumul se adresează proiectanţilor de algoritmi şi cuprinde întregul drum de la teorie la program. Baza teoretică este uşor accesibilă, fiind discutată direct prin prisma conceptelor familiare domeniului programării în locul unei abordări abstracte, pur matematice. Cartea poate fi folosită atât ca text de bază adresat studenţilor ce urmează cursuri de specializare în calculatoare, cât şi drept un ghid util dezvoltatorilor şi programatorilor unor aplicaţii ce impun o contribuţie algoritmică semnificativă. Un ghid de analiză, dezvoltare şi, în unele cazuri, de implementare a algoritmilor convenţionali, aleatori, euristici şi de aproximare, folosind elementele teoretice necesare înţelegerii şi justificării unor asemenea algoritmi. Exemplele sunt focalizate mai ales pe probleme de prelucrare a grafurilor, dată fiind sugestivitatea şi aria largă a aplicabilităţii acestora.

Ferucio Laurenţiu Ţiplea

## Fundamentele algebrice ale informaticii




Lucrarea este concepută ca suport de curs pentru studenţii din primul an ai facultăţilor de informatică din întreaga ţară. Predat de autor la School of Electrical Engineering and Computer Science, University of Central Florida, acesta prezintă elementele de bază necesare înţelegerii unor capitole fundamentale de informatică, împreună cu aplicaţii concrete în următoarele domenii: teoria codurilor şi compresia datelor; teoria codurilor detectoare şi corectoare de erori; criptografie şi securitatea informaţiei; semantica limbajelor de programare; analiza programelor; teoria tipurilor de date.

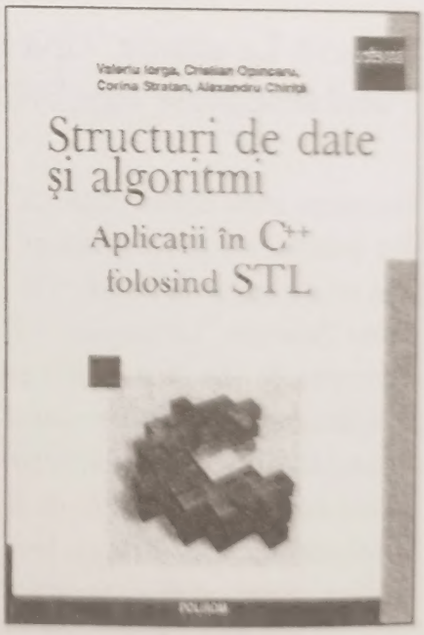
Marin Fotache

## Proiectarea bazelor de date. Normalizare și postnormalizare. Implementări SQL și Oracle

De la megaaplicațiile bancare, industriale sau militare până la banala agendă de telefon, bazele de date au devenit un ingredient indispensabil, chiar dacă mai puțin vizibil, al vieții de zi cu zi. Caracterul „discret", dar riguros conceptualizat și formalizat, la care se adaugă totuși o considerabilă doză de improvizație și intuiție, au făcut din bazele de date un domeniu deosebit de atractiv pentru profesioniștii IT, proiectanții de aplicații și chiar simplii utilizatori. Adresată studenților, administratorilor de baze de date și programatorilor, cartea tratează, la nivel tehnic, conceptual și metodologic, însoțit permanent de exemple practice, problema delicată a proiectării unei baze de date. Practic, lucrarea prezintă cum s-ar putea „gândi" o schemă de baze de date pentru o aplicație anume, ce elemente ar trebui luate în considerare, chestiunile mai riguroase și cele mai flexibile, întregul demers fiind bazat pe normalizare și postnormalizare, precum și pe mecanismul de declarare a restricțiilor de integritate, coerență și consistență a datelor.

Valeriu Iorga, Cristian Opincaru, Corina Stratan, Alexandru Chiriță

## Structuri de date și algoritmi. Aplicații în C++ folosind STL




~nvățarea prin exemple este o metodă larg aplicată în numeroase domenii. Un program clar, concis, comentat, dezvoltat structurat este mai convingător decât un manual de referință voluminos. Lucrarea reprezintă o culegere de probleme rezolvate cu ajutorul bibliotecii de șabloane standard (STL). Programele dezvoltate folosind STL beneficiază de viteze de dezvoltare și de execuție sporite. Ele sunt mai eficiente, mai robuste, mai portabile, mai ușor de modificat și de întreținut decât abordările clasice. Fiecare capitol conține un breviar, în care sunt prezentate principalele noțiuni folosite pe parcurs, un subcapitol de probleme rezolvate, în care se descrie metoda de rezolvare adoptată, se discută complexitatea ei și se prezintă programul sursă testat, un subcapitol de probleme propuse și o bibliografie referitoare la capitolul respectiv.

B.C.U. „M. EMINESCU" IAȘI
MATEMATICĂ

**www.polirom.ro**

Coperta: Radu Răileanu

Bun de tipar: septembrie 2008. Apărut: 2008
Editura Polirom, B-dul Carol I nr. 4 • P.O. Box 266, 700506, Iaşi
Tel. & Fax: (0232) 21.41.00; (0232) 21.41.11;
(0232) 21.74.40 (difuzare); E-mail: office@polirom.ro
B-dul I.C. Brătianu nr. 6, et. 7, ap. 33; O.P. 37 •
P.O. Box 1-728, 030174, Bucureşti;
Tel.: (021) 313.89.78, E-mail: office.bucuresti@polirom.ro

Tiparul executat la S.C. LUMINA TIPO s.r.l.
str. Luigi Galvani nr. 20 bis, sect. 2, Bucureşti
Tel./Fax: 211.32.60, 212.29.27, E-mail: office@luminatipo.com

profesional
fundamente

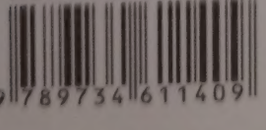
# PROIECTAREA ALGORITMILOR

Gândirea algoritmică presupune deprinderea unui limbaj specific, independent de limbajul de programare, şi abilităţi de formalizare, modelare, abstractizare şi sintetizare. Volumul prezintă, prin intermediul a numeroase exemple, tehnicile de proiectare a algoritmilor şi elementele de analiză a acestora: corectitudinea, timpul de execuţie şi spaţiul de memorie utilizat. Fiecare capitol conţine ilustraţii şi secvenţe de cod, iar la final sunt oferite referinţe bibliografice pentru aprofundarea temei şi exerciţii cu niveluri diferite de dificultate. *Proiectarea algoritmilor* se adresează studenţilor şi elevilor de la facultăţile şi liceele de informatică, precum şi profesorilor şi dezvoltatorilor de software.

**Tipuri de date • Derecursivare • Sortare • Căutare • Algoritmi *greedy* • *Divide-et-impera* • Programare dinamică • *Backtracking* • Probleme NP-complete • *Branch-and-bound***

oto copertă © Tatiana Nikolaevna Kalashnikova/Dreamstime.com

www.polirom.ro
ISBN 978-973-46-1140-9



POLIROM